Exploring Dynamic Compilation and Cross-Layer Object Management Policies for Managed Language Applications

By

Michael Jantz

Submitted to the Department of Electrical Engineering and Computer Science and the
Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Committee members

Prasad A. Kulkarni, Chairperson

Bo Luo

Andrew Gill

Xin Fu

Karen Nordheden

Date defended: _____

The Dissertation Committee for Michael Jantz certifies
that this is the approved version of the following dissertation :

Exploring Dynamic Compilation and Cross-Layer Object Management Policies for Managed Language Applications

Prasad A. Kulkarni, Chairperson

Date approved: _____

Abstract

Recent years have witnessed the widespread adoption of managed programming languages that are designed to execute on virtual machines. Virtual machine architectures provide several powerful software engineering advantages over statically compiled binaries, such as portable program representations, additional safety guarantees, automatic memory and thread management, and dynamic program composition, which have largely driven their success. To support and facilitate the use of these features, virtual machines implement a number of services that adaptively manage and optimize application behavior during execution. Such runtime services often require tradeoffs between efficiency and effectiveness, and different policies can have major implications on the system's performance and energy requirements.

In this work, we extensively explore policies for the two runtime services that are most important for achieving performance and energy efficiency: dynamic (or Just-In-Time (JIT)) compilation and memory management. First, we examine the properties of single-tier and multi-tier JIT compilation policies in order to find strategies that realize the best program performance for existing and future machines. Our analysis performs hundreds of experiments with different compiler aggressiveness and optimization levels to evaluate the performance impact of varying *if* and *when* methods are compiled. We later investigate the issue of *how* to optimize program regions to maximize performance in JIT compilation environments. For this study, we conduct a thorough analysis of the behavior of optimization phases in our dynamic compiler, and construct a custom experimental framework to determine the performance limits of phase selection during dynamic compilation. Next, we explore innovative memory

management strategies to improve energy efficiency in the memory subsystem. We propose and develop a novel *cross-layer* approach to memory management that integrates information and analysis in the VM with fine-grained management of memory resources in the operating system. Using custom as well as standard benchmark workloads, we perform detailed evaluation that demonstrates the energy-saving potential of our approach. We implement and evaluate all of our studies using the industry-standard Oracle HotSpot Java Virtual Machine to ensure that our conclusions are supported by widely-used, state-of-the-art runtime technology.

Acknowledgements

I would like to thank my advisor, Prasad Kulkarni, whose guidance and mentorship has been the most significant force in growing my skills and shaping my attitudes towards scientific research. Without his patience and support, this work would not have been possible. I must also thank my family: my parents, whose love and encouragement helped me get through the hardest and most stressful times of my Ph.D., and my sisters, particularly, Marianne, who served as a sounding board for many of the ideas presented in this dissertation. Finally, I thank my wife, Leslie, whose love, advice, and cooking kept me fueled and motivated to finish this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the introduction of the Java programming language almost two decades ago, applications written in managed languages have become ubiquitous in domains ranging from embedded devices to enterprise servers. A major reason for their popularity is that managed languages have the distinct advantage of program portability; each application is distributed as machine-independent binary codes and executes inside a *virtual machine* (VM) environment (also called runtime system). In addition to enabling portable program emulation, VMs provide a convenient sandbox for monitoring and controlling application behavior at runtime. This architecture allows managed languages to deliver a number of other powerful features, such as garbage collection and dynamic class loading, that enhance the user-end programming model. To support these features, managed languages implement a number of services in the runtime system for managing and optimizing application behavior during execution. Despite their benefits, these services often introduce significant overheads and have created new challenges for achieving high performance.

Therefore, much research and industry effort has been focused on finding techniques and policies to improve the efficiency of runtime services. This technology often relies upon program monitoring, such as online profiling or sampling, to help guide optimization and management decisions. In many cases, virtual machines have to carefully balance tradeoffs between the amount of overhead that can be tolerated and the benefits that can be delivered by the runtime service.

For example, consider the evolution of runtime emulation technology. Since the application binary format does not match the native architecture, VMs have to translate program instructions to machine code at runtime. Unfortunately, simple interpretation schemes incur large overheads and are too slow to compete with native execution. Thus, virtual machines have incorporated dynamic or Just-In-Time (JIT) compilation to improve emulation performance. However, since it occurs at runtime, JIT compilation contributes to the overall execution time of the application and can potentially impede application progress and further degrade its *response* time, if performed injudiciously. To address these issues, researchers invented *selective compilation* techniques to control if and when to compile program methods (Hölzle & Ungar, 1996; Paleczny et al., 2001; Krintz et al., 2000; Arnold et al., 2005). Selective compilation uses online profiling to identify the subset of *hot* methods that the application executes most often, and compiles these methods at program startup, This technique effectively limits the overhead of JIT compilation, while still deriving most of its performance benefits. Most current VMs employ selective compilation with a *staged* emulation model (e.g. interpret first, compile later), and use heuristics to balance throughput performance with response time (Hansen, 1974).

In this work, we explore a wide range of system policies and configurations as well as propose innovative solutions for managing tradeoffs and improving optimization decisions in virtual machines. Our goal is to discover strategies and techniques that improve system performance and energy efficiency for a wide range of managed language applications. Thus, we target our investigation to runtime services that are most likely to impact execution efficiency, specifically, dynamic compilation and memory management. We conduct all of our studies using applications written in Java (today's most popular managed language), and employ the industry-standard Oracle HotSpot Java Virtual Machine (JVM) to implement and evaluate each VM-based approach.

We first investigate the questions of *if*, *when*, and *how* to compile and optimize program regions during execution to maximize program efficiency. Previous research has shown that a conservative JIT compilation policy is most effective to obtain good runtime performance without impeding application progress on single-core machines. However, we observe that the best strategy depends

on architectural features, such as the number of available computing cores, as well as characteristics of the runtime compiler, including the number of compiling threads and how fast and how well methods are optimized. Our first study, presented in the next chapter, explores the properties of single-tier and multi-tier JIT compilation policies that enable VMs to realize the best program performance on modern machines. We design and implement an experimental framework to effectively control aspects of if and when methods are compiled, and perform hundreds of experiments to determine the best strategy for current single/multi-core as well as future many-core architectures.

Our second study, presented in Chapter 3, examines the problem of *phase selection* in dynamic compilers. Customizing the applied set of optimization phases for individual methods or programs has been found to significantly improve the performance of *statically* generated code. However, the performance potential of phase selection for JIT compilers is largely unexplored. For this study, we develop an open-source, production-quality framework for applying program-wide or method-specific customized phase selections using the HotSpot server compiler as base. Using this framework, we conduct novel experiments to understand the behavior of optimization phases relevant to the phase selection problem. We employ long-running genetic algorithms to determine the performance limits of customized phase selections, and later use these results to evaluate existing state-of-the-art heuristics.

In contrast to program emulation technology, which is, in most cases, confined to the virtual machine, most systems provide operating system and hardware support to virtualize applications' access to memory resources. Despite its widespread popularity and native support, virtual memory creates a number of challenges for system optimizers and engineers. Specifically, it is very difficult to obtain precise control over the distribution of memory capacity, bandwidth, and/or power, when virtualizing system memory. For our second set of studies, we propose innovative memory management strategies to overcome these challenges and enable more efficient distribution of memory resources. In Chapter 4, we propose and implement our *application guidance* framework for virtual memory systems. Our approach improves collaboration between the applications, operating

system, and memory hardware (i.e. controllers and DIMMs) during memory management in order to balance power and performance objectives. It includes an application programming interface (API) that enables applications to efficiently communicate different provisioning goals concerning groups of virtual ranges to the kernel. The OS incorporates this information while performing physical page allocation and recycling to achieve the various objectives. In this work, we describe the design and implementation of our application guidance framework, and present examples and experiments to showcase and evaluate the potential of our approach.

Our application guidance framework relies on engineers to manually determine and insert effective memory usage guidance into application source code to realize provisioning goals. Unfortunately, these requirements are infeasible for many workloads and usage scenarios. Thus, for our final study, which is presented in Chapter 5, we integrate our application guidance framework with the HotSpot JVM to provide an *automatic* cross-layer memory management framework. Our implementation develops a novel profiling-based analysis and code generator in HotSpot that automatically classify and organize Java program objects into separate heap regions to improve energy efficiency. We evaluate our framework using a combination of custom as well as standard benchmark workloads and find that it achieves significant DRAM energy savings without requiring any source code modifications or re-compilations.

The studies presented in this dissertation make significant progress towards understanding and improving program efficiency with virtual machine runtime services. We investigate a wide range of strategies and implement a number of new techniques to optimize power and performance for managed language applications. In sum, the major contributions of this dissertation are:

- We conduct a thorough exploration of various factors that affect JIT compilation strategy, and provide policy recommendations for available single/multi-core and future many-core machines,

- We construct a robust, open-source framework for exploring dynamic compiler phase selection in the HotSpot JVM, and, using this framework, present the first analysis of the *ideal* benefits of phase selection in an online JIT compilation environment,

- We design and implement the first-ever virtual memory system to allow applications to provide guidance to the operating system during memory management, and present several examples to showcase and evaluate the benefits of this approach, and
- We integrate our application guidance framework with the HotSpot JVM to provide automatic cross-layer memory management, and perform detailed experimental and performance analysis to demonstrate the energy-saving potential of this approach.

Chapter 2

Exploring Single and Multi-Level JIT Compilation Policy for Modern Machines

Dynamic or Just-in-Time (JIT) compilation is essential to achieve high-performance emulation for programs written in *managed* languages, such as Java and C#. It has been observed that a conservative JIT compilation policy is most effective to obtain good runtime performance without impeding application progress on single-core machines. At the same time, it is often suggested that a more aggressive dynamic compilation strategy may perform best on modern machines that provide abundant computing resources, especially with virtual machines (VM) that are also capable of spawning multiple concurrent compiler threads. However, comprehensive research on the best JIT compilation policy for such modern processors and VMs is currently lacking. The goal of this study is to explore the properties of single-tier and multi-tier JIT compilation policies that can enable existing and future VMs to realize the best program performance on modern machines.

In this chapter, we design novel experiments and implement new VM configurations to effectively control the compiler aggressiveness and optimization levels (*if* and *when* methods are compiled) in the industry-standard Oracle HotSpot Java VM to achieve this goal. We find that the best JIT compilation policy is determined by the nature of the application and the speed and effectiveness of the dynamic compilers. We extend earlier results showing the suitability of conserva-

tive JIT compilation on single-core machines for VMs with multiple concurrent compiler threads. We show that employing the free compilation resources (compiler threads and hardware cores) to aggressively compile *more* program methods quickly reaches a point of diminishing returns. At the same time, we also find that using the free resources to reduce compiler queue backup (compile selected hot methods *early*) significantly benefits program performance, especially for slower (highly-optimizing) JIT compilers. For such compilers, we observe that accurately prioritizing JIT method compiles is crucial to realize the most performance benefit with the smallest hardware budget. Finally, we show that a tiered compilation policy, although complex to implement, greatly alleviates the impact of more and early JIT compilation of programs on modern machines.

2.1 Introduction

To achieve application portability, programs written in *managed* programming languages, such as Java (Gosling et al., 2005) and C# (Microsoft, 2001), are distributed as machine-independent intermediate language binary codes for a *virtual machine* (VM) architecture. Since the program binary format does not match the native architecture, VMs have to employ either interpretation or dynamic compilation for executing the program. Additionally, the overheads inherent during program interpretation make dynamic or Just-in-Time (JIT) compilation essential to achieve high-performance emulation of such programs in a VM (Smith & Nair, 2005).

Since it occurs at runtime, JIT compilation contributes to the overall execution time of the application and can potentially impede application progress and further degrade its *response* time, if performed injudiciously. Therefore, JIT compilation policies need to carefully tune *if*, *when*, and *how* to compile different program regions to achieve the best overall performance. Researchers invented the technique of *selective compilation* to address the issues of *if* and *when* to compile program methods during dynamic compilation (Hölzle & Ungar, 1996; Paleczny et al., 2001; Krintz et al., 2000; Arnold et al., 2005). Additionally, several modern VMs provide multiple optimization levels along with decision logic to control and decide *how* to compile each method. While a *single-*

tier compilation strategy always applies the same set of optimizations to each method, a *multi-tier* policy may compile the same method multiple times at distinct optimization levels during the same program run. The control logic in the VM determines each method's *hotness* level (or how much of the execution time is spent in a method) to decide its compilation level.

Motivation: Due to recent changes and emerging trends in hardware and VM architectures, there is an urgent need for a fresh evaluation of JIT compilation strategies on modern machines. Research on JIT compilation policies has primarily been conducted on single-processor machines and for VMs with a single compiler thread. As a result, existing policies that attempt to improve program efficiency while minimizing application pause times and interference are typically quite conservative. Recent years have witnessed a major paradigm shift in microprocessor design from high-clock frequency single-core machines to processors that now integrate multiple cores on a single chip. These modern architectures allow the possibility of running the compiler thread(s) on a separate core(s) to minimize interference with the application thread. VM developers are also responding to this change in their hardware environment by allowing the VM to simultaneously initiate multiple concurrent compiler threads. Such evolution in the hardware and VM contexts may require radically different JIT compilation policies to achieve the most effective overall program performance.

Objective: The objective of this research is to investigate and recommend JIT compilation strategies to enable the VM to realize the best program performance on existing single/multi-core processors and future many-core machines. We vary the *compilation threshold*, the number of initiated compiler threads, and single and multi-tier compilation strategies to control *if*, *when*, and *how* to detect and compile important program methods. The compilation threshold is a heuristic value that indicates the *hotness* of each method in the program. Thus, more aggressive policies employ a smaller compilation threshold so that more methods become *hot* sooner. We induce progressive increases in the aggressiveness of JIT compilation strategies, and the number of concurrent compiler threads and analyze their effect on program performance. While a single-tier compilation strategy uses a single compiler (and fixed optimization set) for each hot method, a multi-tier compiler

policy typically compiles a hot method with progressively *advanced* (that apply more and better optimizations to potentially produce higher-quality code), but slower, JIT compilers. Our experiments change the different multi-tier hotness thresholds in lock-step to also *partially* control how (optimization level) each method is compiled.[1] Additionally, we design and construct a novel VM configuration to conduct experiments for many-core machines that are not commonly available as yet.

Findings and Contributions: This is the first work to thoroughly explore and evaluate these various compilation parameters and strategies 1) on multi-core and many-core machines and 2) together. We find that the most effective JIT compilation strategy depends on several factors, including: the availability of free computing resources, program features (particularly the ratio of hot program methods), and the compiling speed, quality of generated code, and the method prioritization algorithm used by the compiler(s) employed. In sum, the major contributions of this research are:

1. We design original experiments and VM configurations to investigate the most effective JIT compilation policies for modern processors and VMs with single and multi-level JIT compilation.

2. We quantify the impact of altering 'if', 'when', and one aspect to 'how' methods are compiled on application performance. Our experiments evaluate JVM performance with various settings for compiler aggressiveness and the number of compilation threads, as well as different techniques for prioritizing method compiles, with both single and multi-level JIT compilers.

3. We explain the impact of different JIT compilation strategies on available single/multi-core and future many-core machines.

[1] In contrast to the two components of 'if' and 'when' to compile, the issue of how to compile program regions is much broader and is not unique to dynamic compilation, as can be attested by the presence of multiple optimization levels in GCC, and the wide body of research in profile-driven compilation (Graham et al., 1982; Chang et al., 1991; Arnold et al., 2002; Hazelwood & Grove, 2003) and optimization phase ordering/selection (Whitfield & Soffa, 1997; Haneda et al., 2005a; Cavazos & O'Boyle, 2006; Sanchez et al., 2011) for static and dynamic compilers. Consequently, we only explore one aspect of 'how' to compile methods in this chapter, and provide a more thorough examination of these issues in Chapter 3.

The rest of this chapter is organized as follows. In the next section, we present background information and related work on existing JIT compilation policies. We describe our general experimental setup in Section 2.3. Our experiments exploring different JIT compilation strategies for VMs with multiple compiler threads on single-core machines are described in Section 2.4. In Section 2.5, we present results that explore the most effective JIT compilation policies for multi-core machines. We describe the results of our novel experimental configuration to study compilation policies for future many-core machines in Section 2.6. We explain the impact of prioritizing method compiles, and effect of multiple application threads in Sections 2.7 and 2.8. Finally, we present our conclusions and describe avenues for future work in Sections 2.9 and 2.10 respectively.

2.2 Background and Related Work

Several researchers have explored the effects of conducting compilation at runtime on overall program performance and application pause times. The ParcPlace Smalltalk VM (Deutsch & Schiffman, 1984) followed by the Self-93 VM (Hölzle & Ungar, 1996) pioneered many of the adaptive optimization techniques employed in current VMs, including selective compilation with multiple compiler threads on single-core machines. Aggressive compilation on such machines has the potential of degrading program performance by increasing the compilation time. The technique of selective compilation was invented to address this issue with dynamic compilation (Hölzle & Ungar, 1996; Paleczny et al., 2001; Krintz et al., 2000; Arnold et al., 2005). This technique is based on the observation that most applications spend a large majority of their execution time in a small portion of the code (Knuth, 1971; Bruening & Duesterwald, 2000; Arnold et al., 2005). Selective compilation uses online profiling to detect this subset of *hot* methods to compile at program startup, and thus limits the overhead of JIT compilation while still deriving the most performance benefit. Most current VMs employ selective compilation with a *staged* emulation model (Hansen, 1974). With this model, each method is interpreted or compiled with a fast non-optimizing compiler at program start to improve application response time. Later, the VM determines and selectively

compiles and optimizes only the subset of hot methods to achieve better program performance.

Unfortunately, selecting the hot methods to compile requires *future* program execution information, which is hard to accurately predict (Namjoshi & Kulkarni, 2010). In the absence of any better strategy, most existing JIT compilers employ a simple prediction model that estimates that frequently executed *current* hot methods will also remain hot in the future (Grcevski et al., 2004; Kotzmann et al., 2008; Arnold et al., 2000a). Online profiling is used to detect these current hot methods. The most popular online profiling approaches are based on instrumentation *counters* (Hansen, 1974; Hölzle & Ungar, 1996; Kotzmann et al., 2008), interrupt-timer-based *sampling* (Arnold et al., 2000a), or a combination of the two methods (Grcevski et al., 2004). The method/loop is sent for compilation if the respective method counters exceed a fixed threshold.

Finding the correct threshold value is crucial to achieve good program startup performance in a virtual machine. Setting a higher than ideal compilation threshold may cause the virtual machine to be too conservative in sending methods for compilation, reducing program performance by denying hot methods a chance for optimization. In contrast, a compiler with a very low compilation threshold may compile too many methods, increasing compilation overhead. Therefore, most performance-aware JIT compilers experiment with many different threshold values for each compiler stage to determine the one that achieves the best performance over a large benchmark suite.

Resource constraints force JIT compilation policies to make several tradeoffs. Thus, selective compilation limits the time spent by the compiler at the cost of potentially lower application performance. Additionally, the use of online profiling causes delays in making the compilation decisions at program startup. The first component of this delay is caused by the VM waiting for the method counters to reach the compilation *threshold* before *queuing* it for compilation. The second factor contributing to the compilation delay occurs as each compilation request waits in the compiler queue to be serviced by a free compiler thread. Restricting method compiles and the delay in optimizing hot methods results in poor application startup performance as the program spends more time executing in unoptimized code (Kulkarni et al., 2007a; Krintz, 2003; Gu & Verbrugge,

2008).

Various strategies have been developed to address these delays in JIT compilation at program startup. Researchers have explored the potential of offline profiling and classfile annotation (Krintz & Calder, 2001; Krintz, 2003), early and accurate prediction of hot methods (Namjoshi & Kulkarni, 2010), and online program phase detection (Gu & Verbrugge, 2008) to alleviate the first delay component caused by online profiling. Likewise, researchers have also studied techniques to address the second component of the compilation delay caused by the backup and wait time in the method compilation queue. These techniques include increasing the priority (Sundaresan et al., 2006) and CPU utilization (Kulkarni et al., 2007a; Harris, 1998) of the compiler thread, and providing a priority-queue implementation to reduce the delay for the *hotter* program methods (Arnold et al., 2000b).

However, most of the studies described above have only been targeted for single-core machines. There exist few explorations of JIT compilation issues for multi-core machines. Krintz et al. investigated the impact of background compilation in a separate thread to reduce the overhead of dynamic compilation (Krintz et al., 2000). This technique uses a single compiler thread and employs offline profiling to determine and prioritize hot methods to compile. Kulkarni et al. briefly discuss performing parallel JIT compilation with multiple compiler threads on multi-core machines, but do not provide any experimental results (Kulkarni et al., 2007a). Existing JVMs, such as Sun's HotSpot server VM (Paleczny et al., 2001) and the Azul VM (derived from HotSpot), support multiple compiler threads, but do not present any discussions on ideal compilation strategies for multi-core machines. Prior work by Böhm et al. explores the issue of parallel JIT compilation with a priority queue based dynamic work scheduling strategy in the context of their dynamic binary translator (Böhm et al., 2011). Esmaeilzadeh et al. study the scalability of various Java workloads and their power / performance tradeoffs across several different architectures (Esmaeilzadeh et al., 2011). Our earlier publications explore some aspects of the impact of varying the aggressiveness of dynamic compilation on modern machines for JVMs with multiple compiler threads (Kulkarni & Fuller, 2011; Kulkarni, 2011). This chapter extends our earlier works by (a) providing

more comprehensive results, (b) re-implementing most of the experiments in the latest OpenJDK JVM that provides a state-of-the-art multi-tier compiler and supports improved optimizations, (c) differentiating the results and re-analyzing our observations based on benchmark characteristics, (d) exploring different heuristic priority schemes, and (e) investigating the effects of aggressive compilation and multiple compiler threads on the multi-tiered JIT compilation strategies. Several production-grade Java VMs, including the Oracle HotSpot and IBM J9, now adopt a multi-tier compilation strategy, which make our results with the multi-tiered compiler highly interesting and important.

2.3 Experimental Framework

The research presented in this chapter is performed using Oracle's OpenJDK/HotSpot Java virtual machine (build 1.6.0_25-b06) (Paleczny et al., 2001). The HotSpot VM uses interpretation at program startup. It then employs a counter-based profiling mechanism, and uses the sum of a method's *invocation* and loop *back-edge* counters to detect and promote hot methods for compilation. We call the sum of these counters the *execution count* of the method. Methods/loops are determined to be hot if the corresponding method execution count exceeds a fixed threshold. The HotSpot VM allows the creation of an arbitrary number of compiler threads, as specified on the command-line.

The HotSpot VM implements two distinct optimizing compilers to improve application performance beyond interpretation. The *client compiler* provides relatively fast compilation times with smaller program performance gains to reduce application startup time (especially, on single-core machines). The *server compiler* applies an aggressive optimization strategy to maximize performance benefits for longer running applications. We conducted experiments to compare the overhead and effectiveness of HotSpot's client and server compiler configurations. We found that the client compiler is immensely fast, and only *requires about 2% of the time, on average, taken by the server compiler* to compile the same set of hot methods. At the same time, the simple and fast *client compiler is able to obtain most (95%) of the performance gain (relative to interpreted*

code) realized by the server compiler.

In addition to the single-level client and server compilers, HotSpot provides a *tiered compiler* configuration that utilizes and combines the benefits of the client and server compilers. In the most common path in the tiered compiler, each hot method is first compiled with the client compiler (possibly with additional profiling code inserted), and later, if the method remains hot, is recompiled with the server compiler. Each compiler thread in the HotSpot tiered compiler is dedicated to either the client or server compiler, and *each compiler is allocated at least one thread*. To account for the longer compilation times needed by the server compiler, HotSpot automatically assigns the compiler threads at a 2:1 ratio in favor of the server compiler. The property of the client compiler to quickly produce high-quality optimized code greatly influences the behavior of the tiered compiler under varying compilation loads, as our later experiments in this chapter will reveal.

There is a single *compiler queue* designated to each (client and server) compiler in the tiered configuration. These queues employ a simple execution count based priority heuristic to ensure the most active methods are compiled earlier. This heuristic computes the execution count of each method in the appropriate queue since the last queue removal to find the most active method. As the load on the compiler threads increases, HotSpot dynamically increases its compilation thresholds to prevent either the client or server compiler queues from growing prohibitively long. In addition, the HotSpot tiered compiler has logic to automatically remove *stale* methods that have stayed in the queue for too long. For our present experiments, we disable the automatic throttling of compilation thresholds and removal of stale methods to appropriately model the behavior of a generic tiered compilation policy. The tiered compiler uses different thresholds that move in lockstep to tune the aggressiveness of its component client and server compilers. Table 2.1 describes these compilation thresholds and their default values for each compiler in the tiered configuration.

The experiments in this chapter were conducted using all the benchmarks from three different benchmark suites, SPECjvm98 (SPEC98, 1998), SPECjvm2008 (SPEC2008, 2008) and DaCapo-9.12-bach (Blackburn et al., 2006). We employ two inputs (10 and 100) for benchmarks in the SPECjvm98 suite, two inputs (small and default) for the DaCapo benchmarks, and a single input

Table 2.1: Threshold parameters in the tiered compiler

Parameter	Description	Client Default	Server Default
Invocation Threshold	Compile method if invocation count exceeds this threshold	200	5000
Backedge Threshold	OSR compile method if backedge count exceeds this threshold	7000	40000
Compile Threshold	Compile method if invocation + backedge count exceeds this threshold (and invocation count > Minimum Invocation Threshold)	2000	15000
Minimum Invocation Threshold	Minimum number of invocations required before method can be considered for compilation	100	600

(startup) for benchmarks in the SPECjvm2008 suite, resulting in 57 benchmark/input pairs. Two benchmarks from the DaCapo benchmark suite, *tradebeans* and *tradesoap*, did not always run correctly with the *default* version of the HotSpot VM, so these benchmarks were excluded from our set. In order to limit possible sources of variation in our experiments, we set the number of application threads to one whenever possible. Unfortunately, several of our benchmarks employ multiple application threads due to *internal multithreading* that cannot be controlled by the harness application. Table 2.2 lists the name, number of invoked methods (under the column labeled #M), and number of application threads (under the column labeled #AT) for each benchmark in our suite.

All our experiments were performed on a cluster of dual quad-core, 64-bit, x86 machines running Red Hat Enterprise Linux 5 as the operating system. The cluster includes three models of server machine: Dell M600 (two 2.83GHz Intel Xeon E5440 processors, 16GB DDR2 SDRAM), Dell M605 (two 2.4GHz AMD Opteron 2378 processors, 16GB DDR2 SDRAM), and PowerEdge SC1435 (two 2.5GHz AMD Opteron 2380 processors, 8GB DDR2 SDRAM). We run all of our experiments on one of these three models, but experiments comparing runs of the same benchmark always use the same model. There are no hyperthreading or frequency scaling techniques of any kind enabled during our experiments.

We disable seven of the eight available cores to run our single-core experiments. Our multi-core experiments utilize all available cores. More specific variations made to the hardware configuration are explained in the respective sections. Each benchmark is run in isolation to prevent interference from other user programs. In order to account for inherent timing variations during the benchmark runs, all the performance results in this chapter report the average over 10 runs for each benchmark-

Table 2.2: Benchmarks used in our experiments.

SPECjvm98			SPECjvm2008			DaCapo-9.12-bach		
Name	*#M*	*#AT*	Name	*#M*	*#AT*	Name	*#M*	*#AT*
_201_compress_100	517	1	compiler.compiler	3195	1	avrora_default	1849	6
_201_compress_10	514	1	compiler.sunflow	3082	1	avrora_small	1844	3
_202_jess_100	778	1	compress	960	1	batik_default	4366	1
_202_jess_10	759	1	crypto.aes	1186	1	batik_small	3747	1
_205_raytrace_100	657	1	crypto.rsa	960	1	eclipse_default	11145	5
_205_raytrace_10	639	1	crypto.signverify	1042	1	eclipse_small	5461	3
_209_db_100	512	1	derby	6579	1	fop_default	4245	1
_209_db_10	515	1	mpegaudio	959	1	fop_small	4601	2
_213_javac_100	1239	1	scimark.fft.small	859	1	h2_default	2154	3
_213_javac_10	1211	1	scimark.lu.small	735	1	h2_small	2142	3
_222_mpegaudio_100	659	1	scimark.monte_carlo	707	1	jython_default	3547	1
_222_mpegaudio_10	674	1	scimark.sor.small	715	1	jython_small	2070	2
_227_mtrt_100	658	2	scimark.sparse.small	717	1	luindex_default	1689	2
_227_mtrt_10	666	2	serial	1121	1	luindex_small	1425	1
_228_jack_100	736	1	sunflow	2015	5	lusearch_default	1192	1
_228_jack_10	734	1	xml.transform	2592	1	lusearch_small	1303	2
			xml.validation	1794	1	pmd_default	3881	8
						pmd_small	3058	3
						sunflow_default	1874	2
						sunflow_small	1826	2
						tomcat_default	9286	6
						tomcat_small	9189	6
						xalan_default	2296	1
						xalan_small	2277	1

configuration pair. The *harness* of all our benchmark suites allows each benchmark to be *iterated* multiple times in the same VM run. We measure the performance of each benchmark as the time it takes to invoke and complete one benchmark iteration. Thus, for all of the experiments in the subsequent sections of this paper, any compilation that occurs is performed concurrently with the running application.

However, we first conduct experiments to illustrate the *compilation* and *steady-state* execution time differences of the HotSpot server and client compilers for all our benchmark programs. Figures 2.1 and 2.2 show the results of these experiments. Each experiment employs one of either the server or client compiler. Regardless of which compiler is used, we use the default server compiler threshold to identify hot methods to ensure the same set of methods are compiled in our comparisons. Each steady-state run disables background compilation to enable all hot methods to be compiled in the first program iteration. We reset the method execution counts after each iteration to prevent any other methods from becoming hot in later program iterations. We then allow each benchmark to iterate 11 more times and record the median runtime of these iterations as the time of the steady-state run. For each benchmark, Figure 2.1 plots the ratio of compilation time regis-

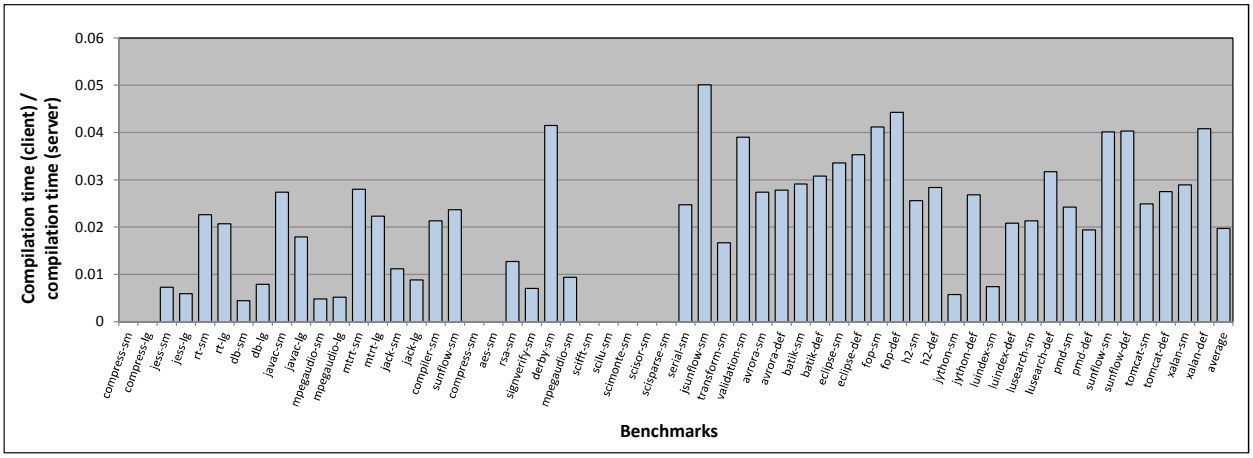


Figure 2.1: Ratio of client and sever compile times when compiling the same number of program methods

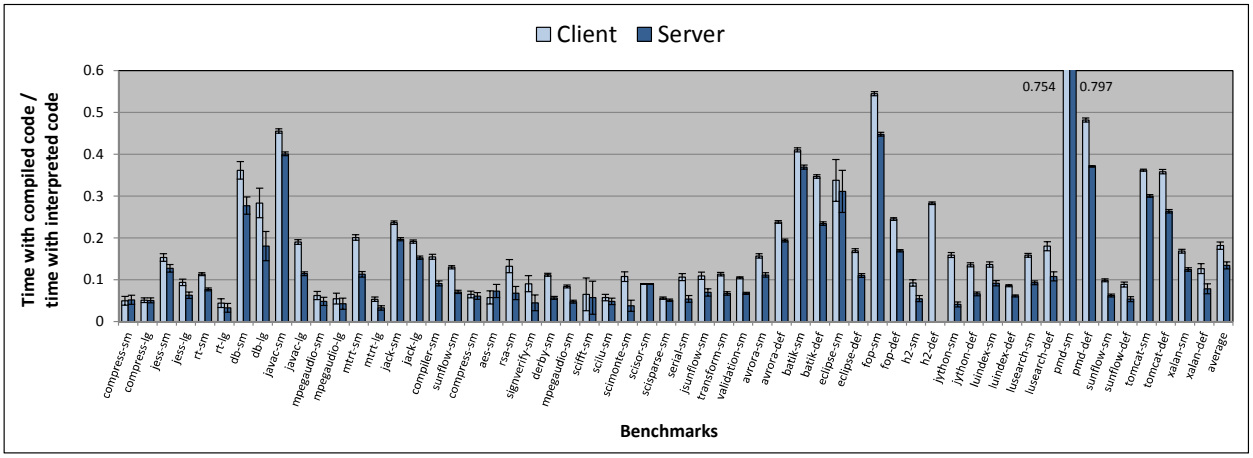


Figure 2.2: Steady-state program execution times using the server and client compilers as a ratio of the interpreted program run-time

tered by the client and server compilers, averaged over 10 steady-state program runs. Our Linux system measures thread times in *jiffies*. The duration of each jiffy, which is configurable at kernel compile time, is about 1msec on our system. Some benchmarks compile only a few methods in our steady-state experiments (the minimum is *monte_carlo* which has only two hot methods with the default server compiler threshold), and require very little compilation time. Cases in which the steady-state run registers less than one jiffy of compilation thread time are reported as zero in Figure 2.1. As we can see, the client compiler is immensely fast, and only *requires about 2% of the time, on average, taken by the server compiler* to compile the same set of methods.

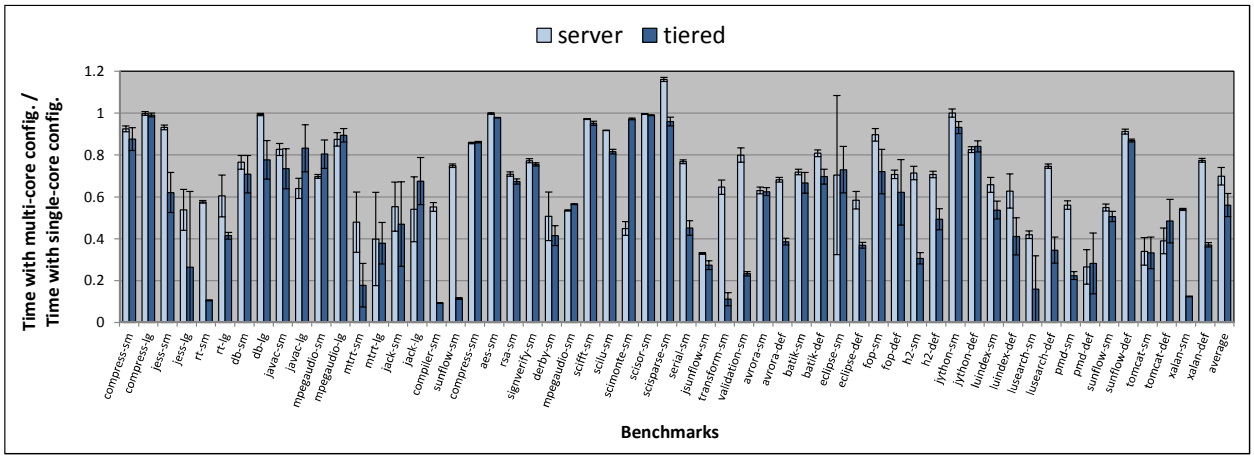


Figure 2.3: Ratio of multi-core performance to single-core performance for each compiler configuration.

Figure 2.2 plots the steady-state benchmark run-time using the client and server compilers as compared with the time required for interpreted execution. To estimate the degree of variability in our run-time results, we compute 95% confidence intervals for the difference between the means as described in (Georges et al., 2007) and plot these intervals as error bars.[2] Figure 2 shows that both the compilers achieve significant program speedups. However, it is interesting to note that the simple and fast *client compiler is able to obtain most of the performance gain realized by the server compiler.* This property of the client compiler to quickly produce high-quality optimized code greatly influences the behavior of the tiered compiler under varying compilation loads, as our later experiments in this chapter will reveal.

Finally, we present a study to compare the program performance on single-core and multi-core machines. Figure 2.3 shows the multi-core performance of each benchmark relative to single-core performance for both the default server and tiered compiler configurations. Not surprisingly, we observe that most benchmarks run much faster with the multi-core configuration. Much of this difference is simply due to increased parallelism, but other micro-architectural effects, such as cache affinity and inter-core communication, may also impact performance depending on the workload. Another significant factor, which we encounter in our experiments throughout this

[2]It is difficult to estimate confidence intervals for the average across all the benchmarks (which is reported as an average of ratios). For these estimates, we assume the maximum difference between the means for each benchmark (in either the positive or negative direction) and re-calculate the average.

work, is that additional cores enable *earlier* compilation of hot methods. This effect accounts for the result that the tiered VM, with its much more aggressive compilation threshold, exhibits a more pronounced performance improvement, on average, than the server VM. The remainder of this chapter explores and explains the impact of different JIT compilation strategies on modern and future architectures using the HotSpot server and tiered compiler configurations.

2.4 JIT Compilation on Single-Core Machines

In this section we report the results of our experiments conducted on single-core processors to understand the impact of aggressive JIT compilation and more compiler threads in a VM on program performance. Our experimental setup controls the aggressiveness of distinct JIT compilation policies by varying the selective compilation threshold. Changing the compilation threshold can affect program performance in two ways: (a) by compiling a lesser or greater percentage of the program code (*if* a method is compiled), and (b) by sending methods to compile early or late (*when* is each method compiled). We first employ the HotSpot server VM with a single compiler thread to find the selective compilation threshold that achieves the best average performance with our set of benchmark programs.[3] Next, we evaluate the impact of multiple compiler threads on program performance for machines with a single processor with both the server and tiered compilers in the HotSpot JVM.

2.4.1 Compilation Threshold with Single Compiler Thread

By virtue of sharing the same computation resources, the application and compiler threads share a complex relationship in a VM running on a single-core machine. A highly selective compile threshold may achieve poor overall program performance by spending too much time executing in non-optimized code resulting in poor overall program run-time. By contrast, a lower than ideal compile threshold may also produce poor performance by spending too long in the compiler thread.

[3]The tiered compiler spawns a minimum of two compiler threads, and, is therefore not used in this single compiler thread configuration.

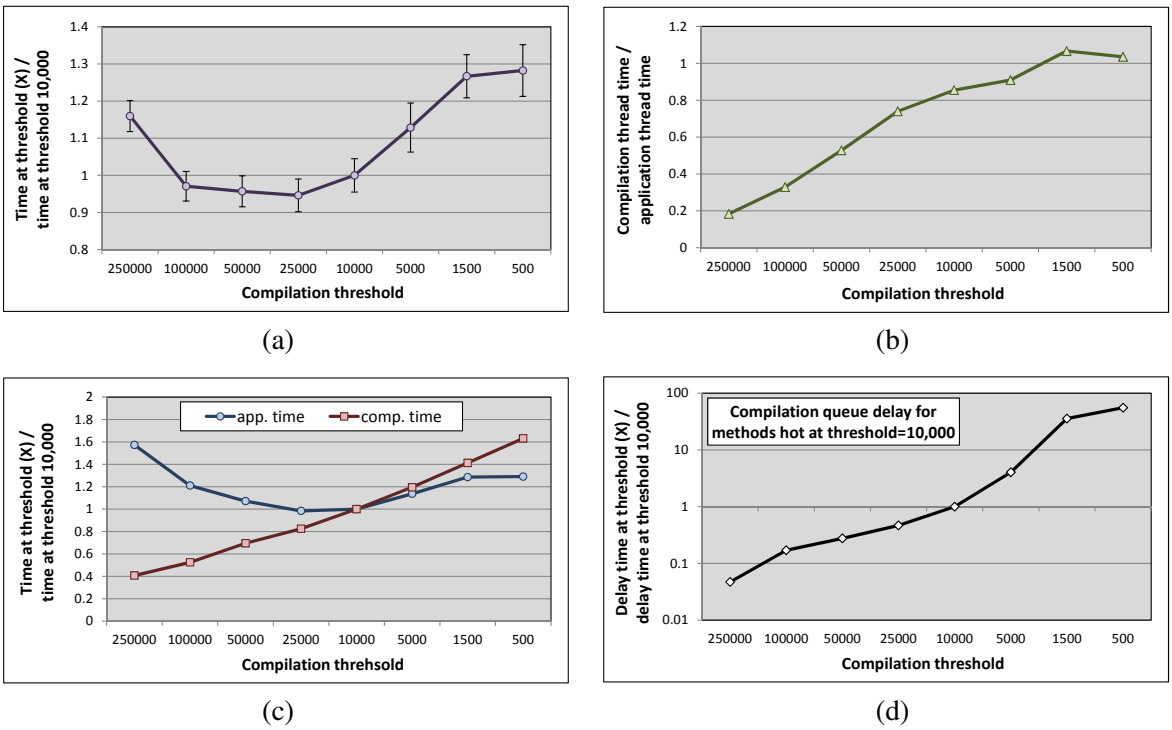


Figure 2.4: Effect of different compilation thresholds on average benchmark performance on single-core processors.

Therefore, the compiler thresholds need to be carefully tuned to achieve the most efficient average program execution on single-core machines over several benchmarks.

We perform an experiment to determine the ideal compilation threshold for the HotSpot server VM with a *single* compiler thread on our set of benchmarks. These results are presented in Figure 2.4(a). The figure compares the average overall program performance at different compile thresholds to the average program performance at the threshold of 10,000, which is the default compilation threshold for the HotSpot server compiler. We find that a few of the less aggressive thresholds are slightly faster, on average, than the default for our set of benchmark programs (although the difference is within the margin of error). The default HotSpot server VM employs two compiler threads and may have been tuned with applications that run longer than our benchmarks, which may explain this result. The average benchmark performance worsens at both high and low compile thresholds.

To better interpret these results, we collect individual thread times during each experiment

to estimate the amount of time spent doing compilation compared to the amount of time spent executing the application. Figure 2.4(b) shows the ratio of compilation to application thread times at each threshold averaged over all the benchmarks. Thus, compilation thresholds that achieve good performance spend a significant portion of their overall runtime doing compilation. We can also see that reducing the compilation threshold increases the relative amount of time spent doing compilation. However, it is not clear how much of this trend is due to longer compilation thread times (from compiling more methods) or reduced application thread times (from executing more native code).

Therefore, we also consider the effect of compilation aggressiveness on each component separately. Figure 2.4(c) shows the break-down of the overall program execution in terms of the application and compiler thread times at different thresholds to their respective times at the compile threshold of 10,000, averaged over all benchmark programs. We observe that high thresholds ($>$ 10,000) compile less and degrade performance by not providing an opportunity to the VM to compile several important program methods. In contrast, the compiler thread times increase with lower compilation thresholds ($<$ 10,000) as more methods are sent for compilation. We expected this increased compilation to improve application thread performance. However, the behavior of the application thread times at low compile thresholds is less intuitive.

On further analysis we found that JIT compilation policies with lower thresholds send more methods to compile and contribute to compiler queue backup. We hypothesize that the flood of less important program methods delays the compilation of the most critical methods, resulting in the non-intuitive degradation in application performance at lower thresholds. To verify this hypothesis we conduct a separate set of experiments that *measure the average compilation queue delay (time spent waiting in the compile queue) of hot methods in our benchmarks.* These experiments compute the mean average compilation queue delay only for methods that are hot at the default threshold of 10,000 for each benchmark / compile threshold combination.

Figure 2.4(d) plots the average compilation queue delay at each compile threshold relative to the average compilation queue delay at the default threshold of 10,000 averaged over the bench-

marks.[4] As we can see, the average compilation queue delay for hot methods increases dramatically as the compilation threshold is reduced. Thus, we conclude that increasing compiler aggressiveness is not likely to improve VM performance running with a single compiler thread on single-core machines.

2.4.2 Effect of Multiple Compiler Threads on Single-Core Machines

In this section we analyze the effect of multiple compiler threads on program performance on a single-core machine with the server and tiered compiler configurations of the HotSpot VM.

2.4.2.1 Single-Core Compilation Policy with the HotSpot Server Compiler

For each compilation threshold, a separate plot in Figure 2.5(a) compares the average overall program performance with multiple compiler threads to the average performance with a single compiler thread at that same threshold. Intuitively, a greater number of compiler threads should be able to reduce the method compilation queue delay. Indeed, we notice program performance improvements for one or two extra compiler threads, but the benefits do not hold with increasing number of such threads ($>$3). We further analyzed the performance degradation with more compiler threads and noticed an increase in the overall *compiler thread* times in these cases. This increase suggests that several methods that were queued for compilation, but never got compiled before program termination with a single compiler thread are now compiled as we provide more VM compiler resources. While the increased compiler activity increases compilation overhead, many of these methods contribute little to improving program performance. Consequently, the potential improvement in application performance achieved by more compilations seems unable to recover the additional compiler overhead, resulting in a net loss in overall program performance.

Figure 2.5(b) compares the average overall program performance in each case to the average performance of a baseline configuration with a single compiler thread at a threshold of 10,000.

[4]We cannot compute a meaningful ratio for benchmarks with zero or very close to zero average compilation queue delay at the baseline threshold. Thus, these results do not include 14 (of 57) benchmarks with an average compilation queue delay less than 1msec (the precision of our timer) at the default threshold.

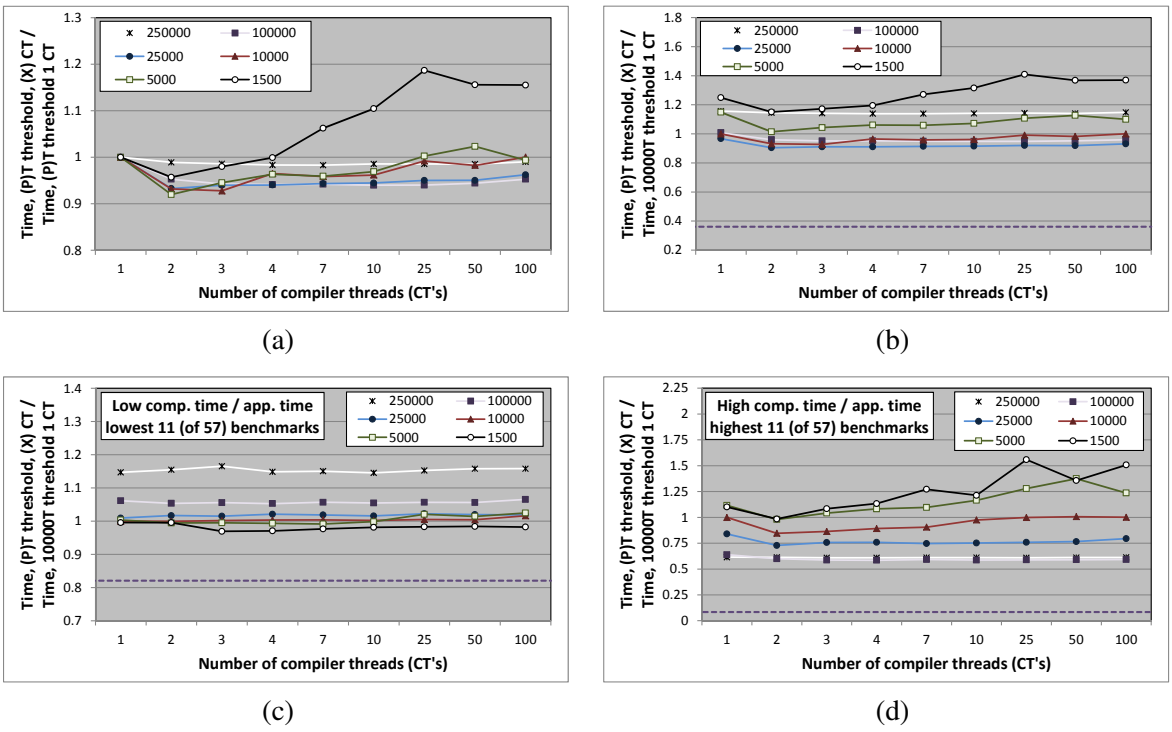


Figure 2.5: Effect of multiple compiler threads on single-core program performance in the HotSpot VM with server compiler. The discrete measured thread points are plotted equi-distantly on the x-axis.

These results reveal the best compiler policy on single-core machines with multiple compiler threads. Thus, we can see that, on average, the more aggressive thresholds perform quite poorly, while moderately conservative thresholds fare the best (with any number of compiler threads). Our analysis finds higher compiler aggressiveness to send more program methods for compilation, which includes methods that may not make substantial contributions to performance improvement (*cold* methods). Additionally, the default server compiler in HotSpot uses a simple FIFO (first-in first-out) compilation queue, and compiles methods in the same order in which they are sent. Consequently, the cold methods delay the compilation of the really important hot methods relative to the application thread, producing the resultant loss in performance.

To further evaluate the configurations with varying compilation resources and aggressiveness (in these and later experiments), we design an *optimal* scenario that measures the performance of each benchmark *with all of its methods pre-compiled*. Thus, the 'optimal' configuration reveals the best-case benefit of JIT compilation. The dashed line in Figure 2.5(b) shows the optimal run-

time on the single-core machine configuration relative to the same baseline startup performance (single benchmark iteration with one compiler thread and a threshold of 10,000), averaged over all the benchmarks. Thus, the "optimal" steady-state configuration achieves much better performance compared to the "startup" runs that compile methods concurrently with the running application on single-core machines. On average, the optimal performance is about 64% faster than the baseline configuration and about 54% faster than the fastest compilation thread / compile threshold configuration (with two compilation threads and a compile threshold of 25,000).

Figure 2.5(c) shows the same plots as in Figure 2.5(b) but only for the 11 (20%) benchmarks with the lowest compilation to application time ratio. Thus, for applications that spend relatively little time compiling, only the very aggressive compilation thresholds cause some compilation queue delay and may produce small performance improvements in some cases. For such benchmarks, all the hot methods are always compiled before program termination. Consequently, the small performance improvements with the more aggressive thresholds are due to compiling hot methods earlier (reduced queue delay). Furthermore, there is only a small performance difference between the startup and optimal runs. By contrast, Figure 2.5(d) only includes the 11 (20%) benchmarks with a relatively high compilation to application time ratio. For programs with such high compilation activity, the effect of compilation queue delay is more pronounced. We find that the less aggressive compiler policies produce better efficiency gains for these programs, but there is still much room for improvement as evidenced by optimal performance results.

These observations suggest that a VM that can adapt its compilation threshold based on the compiler load may achieve the best performance for all programs on single-core machines. Additionally, implementing a priority-queue to order compilations may also enable the more aggressive compilation thresholds to achieve better performance. We explore the effect of prioritized method compiles on program performance in further detail in Section 2.7. Finally, a small increase in the number of compiler threads can also improve performance by reducing the compilation queue delay.

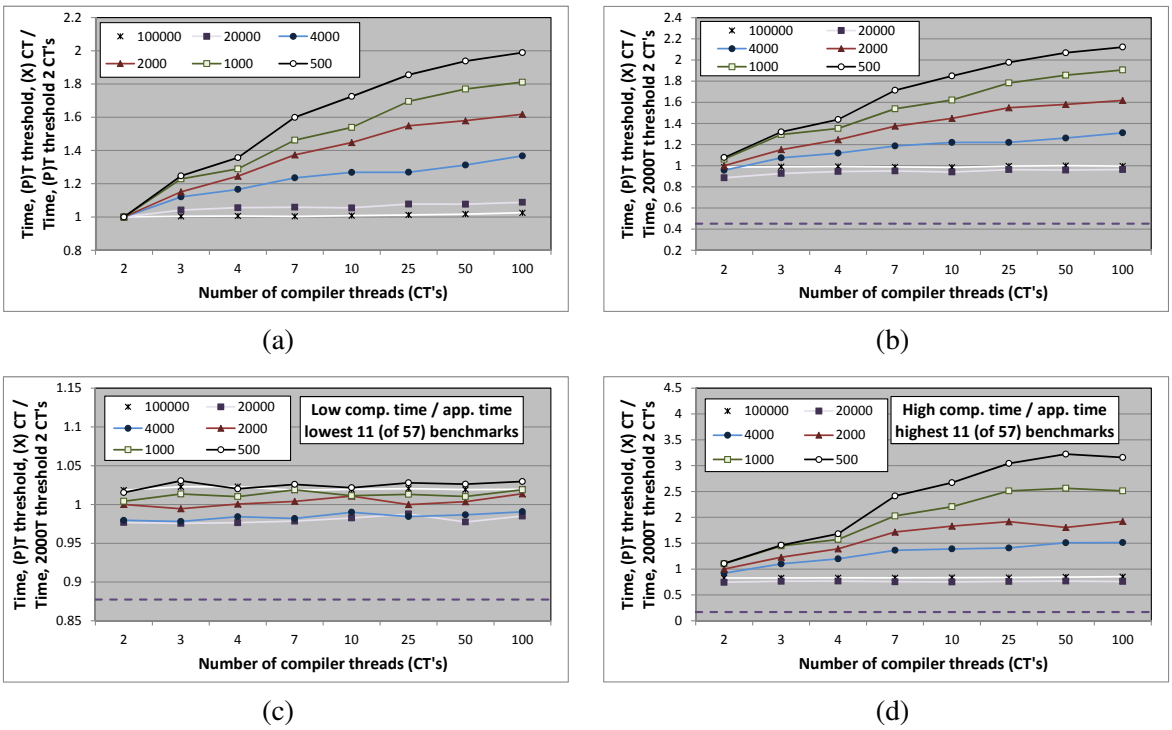


Figure 2.6: Effect of multiple compiler threads on single-core program performance in the HotSpot VM with tiered compiler. The discrete measured thread points are plotted equi-distantly on the x-axis.

2.4.2.2 Single-Core Compilation Policy with the HotSpot Tiered Compiler

In this section we explore the effect on program performance of changing the compiler aggressiveness and the number of compiler threads with a tiered compiler configuration on single-core machines. For our experiments with the tiered compiler, we vary the client and server compiler thresholds in lock-step to adjust the aggressiveness of the tiered compiler and use the corresponding first-level (client) compiler threshold in the graph legends.

Each line-plot in Figure 2.6(a) compares the average overall program performance in the tiered compiler with multiple compiler threads to the average performance with only one client and one server compiler thread at that same threshold. In contrast to the server compiler configuration, increasing the number of compiler threads does not yield any performance benefit and, for larger increases, significantly degrades performance at every threshold. This effect is likely due to the combination of two factors: (a) a very fast first-level compiler that prevents significant backup in

its compiler queue even with a single compiler thread while achieving most of the performance benefits of later re-compilations, and (b) the priority heuristic used by the tiered compiler that may be able to find and compile the most important methods first. Thus, any additional compilations performed by more compiler threads only increase the compilation overhead without commensurately contributing to program performance. In Section 2.7.2, we compare the default tiered compiler to one which employs FIFO (first-in first-out) compilation queues to evaluate the effect of prioritized compilation queues on program performance.

Figure 2.6(b) compares the average program performances in each case to the average performance of the baseline tiered configuration with one client and one server compiler thread and the default threshold parameters (with a client compiler threshold of 2,000). The default tiered compiler employs significantly more aggressive compilation thresholds compared to the default stand-alone server compiler, and, on average, queues up more than three times as many methods for compilation. Consequently, relatively conservative compile thresholds achieve the best performance on single-core machines. The dashed line in Figure 2.6(b) plots the runtime of the optimal configuration (measured as described in the previous section) relative to the runtime of the baseline tiered configuration. Thus, with the tiered VM on single-core machines, the optimal run-time is still much faster than any other start-up configuration. However, due to the fast client compiler and effective priority heuristic, the performance of the tiered VM is significantly closer (10% in the best case) to the optimal runtime than the server VM configurations presented in the previous section.

We again observe that applications with extreme (very low or very high) compilation activity show different performance trends than the average over the complete set of benchmarks. Figure 2.6(c) plots the average performance of the HotSpot tiered compiler VM at different threshold and compiler thread configurations for the 20% benchmarks with the lowest compilation to application time ratio. As expected, compile threshold aggressiveness and the amount of compilation resources have much less of a performance impact on these applications. Additionally, the performance achieved is much closer to the optimal runtime for this set of benchmarks. However, in con-

trast to the server compiler results in Figure 2.5(c), some less aggressive thresholds are *marginally* more effective in the tiered compiler, which again indicates that the compilation queue delay is much less of a factor in the presence of a fast compiler and a good heuristic for prioritizing method compiles. Alternatively, in Figure 2.6(d), we study benchmarks with relatively high compilation activity, and find that less aggressive compile thresholds yield very significant performance gains, due to a very aggressive default threshold used by the tiered compiler.

In summary, for single-core machines it is crucial to select the compiler threshold such that only the most dominant program methods are sent to compilation. With such an ideal compiler threshold, only two compiler threads (one client and one server) are able to service all compilation requests prior to program termination. An ideal threshold combined with a very fast client compiler and a good priority heuristic negates any benefit of additional compiler threads reducing the queue delay. A less aggressive threshold lowers program performance by not allowing the tiered VM to compile all hot methods. In contrast, with more aggressive thresholds, a minimum number of compiler threads are not able to service all queued methods, producing performance degradations due to the overhead of increased compiler activity with more compiler threads.

2.5 JIT Compilation on Multi-Core Machines

Dynamic JIT compilation on single-processor machines has to be conservative to manage the compilation overhead at runtime. Modern multi-core machines provide the opportunity to spawn multiple compiler threads and run them concurrently on separate (free) processor cores, while not interrupting the application thread(s). As such, it is a common perception that a more aggressive compilation policy is likely to achieve better application thread and overall program performance on multi-core machines for VMs with multiple compiler threads. Aggressiveness, in this context, can imply compiling early or compiling more methods by lowering the compile threshold. In this section, we report the impact of varying JIT compilation aggressiveness and the number of compiler threads on the performance of the server and tiered VM on multi-core machines.

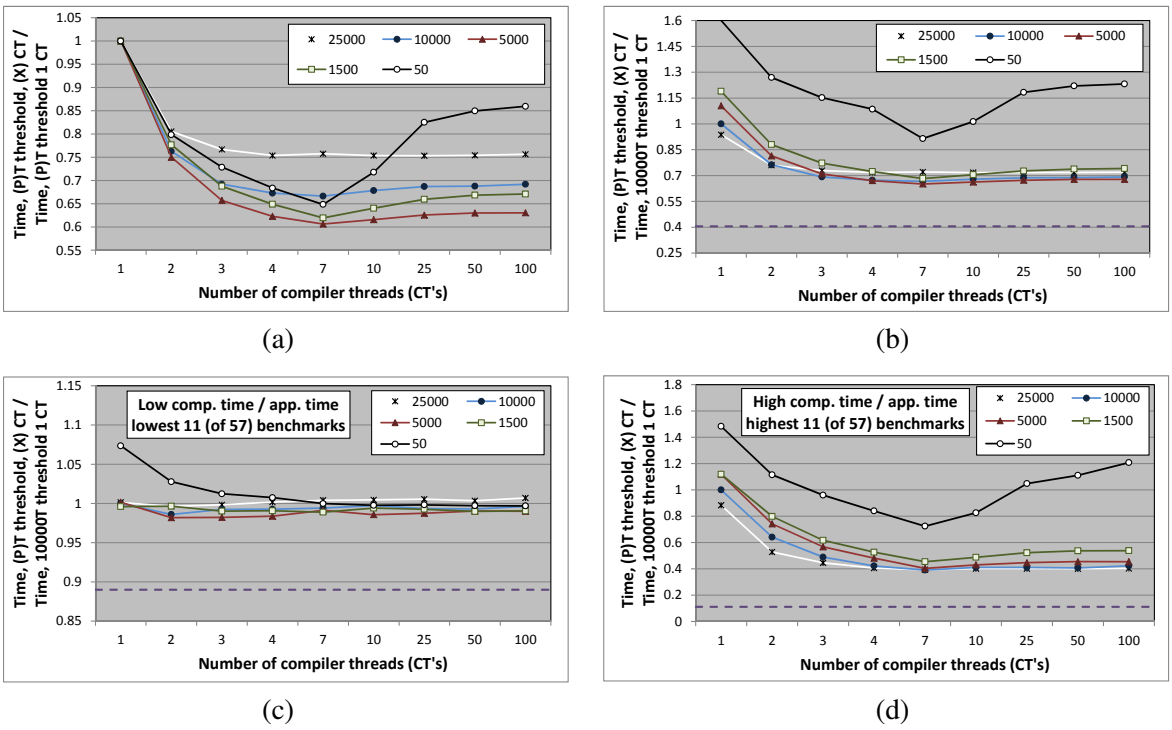


Figure 2.7: Effect of multiple compiler threads on multi-core application performance with the HotSpot Server VM

2.5.1 Multi-Core Compilation Policy with the HotSpot Server Compiler

Figure 2.7 illustrates the results of our experiments with the HotSpot server compiler on multi-core machines. For each indicated compile threshold, a corresponding line-plot in Figure 2.7(a) shows the ratio of the program performance with different number of compiler threads to the program performance with a single compiler thread at that same threshold, averaged over our 57 benchmark-input pairs. Thus, we can see that increasing the number of compiler threads up to seven threads improves application performance at all compile thresholds. However, larger increases in the number of compiler threads (> 7) derive no performance benefits and actually degrade performance with the more aggressive compilation thresholds.

As mentioned earlier, additional compiler threads can improve performance by reducing compilation queue delay, allowing the important program methods to be compiled earlier. Early compilation allows a greater fraction of the program execution to occur in optimized native code (rather than being interpreted), which produces significant gains in program performance. The additional

compiler threads impose minimal impediment to the application threads as long as that computation can be off-loaded onto free (separate) cores. Our existing hardware setup only provides eight distinct processing cores. Consequently, larger increases in the number of compiler threads cause application and compilation threads to compete for machine resources. Moreover, configurations with aggressive compilation thresholds frequently compile methods that derive little performance benefit. This additional (but incommensurate) compilation overhead can only be sustained as long as compilation is free, and results in significant performance losses in the absence of free computational resources.

Figure 2.7(b) compares all the program performances (with different thresholds and different number of compiler threads) to a single baseline program performance. The selected baseline is the program performance with a single compiler thread at the default HotSpot server compiler threshold of 10,000. We can see that while the optimal performance (again indicated by the dashed line) is much faster than the performance of the baseline configuration, increasing compilation activity on otherwise free compute cores enables the server VM to make up much of this difference. In the best case (configuration with threshold 5,000 and 7 compiler threads) the combination of increased compiler aggressiveness with more compiler threads improves performance by 34.6%, on average, over the baseline. However, most of that improvement (roughly 33%) is obtained by simply reducing the compilation queue delay that is realized by increasing the number of compiler threads at the default HotSpot (10,000) threshold. Thus, the higher compiler aggressiveness achieved by lowering the selective compilation threshold seems to offer relatively small benefits over the more conservative compiler policies.

Another interesting observation that can be made from the plots in Figure 2.7(b) is that aggressive compilation policies require more compiler threads (implying greater computational resources) to achieve *good* program performance. Indeed, our most aggressive threshold of 50 performs extremely poorly compared to the default threshold with only one compiler thread (over 60% worse), and requires seven compiler threads to surpass the baseline performance.

As the compiler thresholds get more aggressive, we witness (from Figure 2.7(a)) successively

larger performance losses with increasing the number of compiler threads beyond seven. These losses are due to increasing application interference caused by compiler activity at aggressive thresholds and are a result of the computational limitations in the available hardware. In Section 2.6 we construct a simulation configuration to study the behavior of aggressive compilation policies with large number of compiler threads on (many-core) machines with virtually unlimited computation resources.

Similar to our single-core configurations, we find that these results change dramatically depending on the compilation characteristics of individual benchmark applications. Figures 2.7(c) and 2.7(d) plot the average performance at each multi-core compilation threshold and compiler thread configurations for 20% of the benchmarks with the lowest and highest compilation to application time ratios in our baseline configuration respectively. Varying compilation thresholds and resources has much less of a performance effect on benchmarks that spend relatively little time doing compilation. The best configuration for these benchmarks (with a compilation threshold of 5000 and two compiler threads) yields less than a 2% improvement over the baseline configuration. Also, for these benchmarks, the baseline configuration achieves performance that is much closer to optimal (again, indicated by the dashed line) compared to the overall average in Figure 2.7(b). Alternatively, for benchmarks that spend relatively more time doing compilation, as shown in Figure 2.7(d), there is even more room for improvement compared to the average over all benchmarks. As expected, exploiting the free processor resources to spawn additional compiler threads results in a more substantial performance benefit (an average efficiency gain of over 60%) for these 11 benchmarks.

2.5.2 Multi-Core Compilation Policy with the HotSpot Tiered Compiler

In this section we evaluate the effect of varying compiler aggressiveness on the overall program performance delivered by the VM with its tiered compilation policy. The compilation thresholds for the two compilers in our tiered experimental configurations are varied in lock-step so that they always maintain the same ratio. For each compilation threshold setting with the tiered compiler,

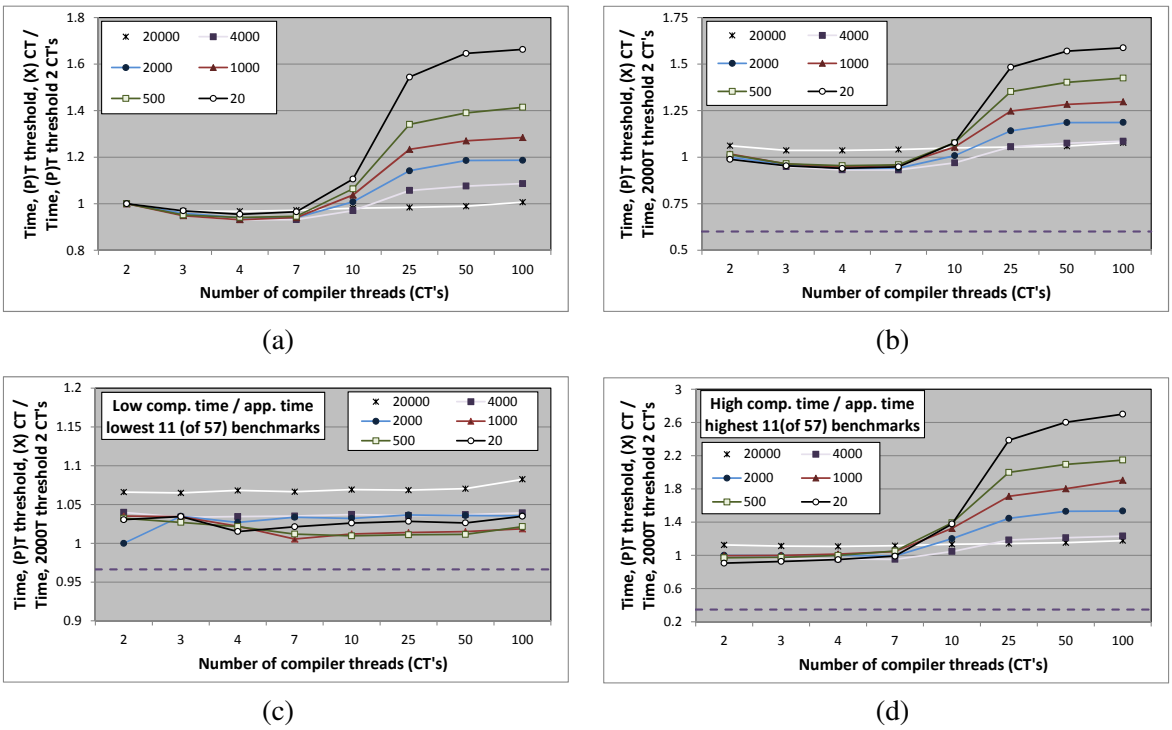
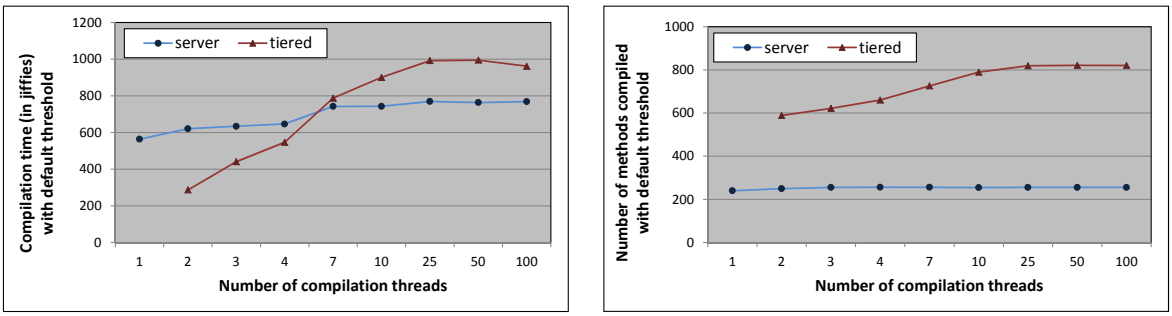


Figure 2.8: Effect of multiple compiler threads on multi-core application performance with the HotSpot Tiered VM

a separate plot in Figure 2.8(a) compares the average overall program performance with multiple compiler threads to the average performance with two (one client and one server) compiler threads at that same compilation threshold. In stark contrast to our results in Section 2.5.1 with the server compiler, increasing the number of compiler threads for the tiered compiler only marginally improves performance at any compile threshold. This result is due to the speed and effectiveness of the HotSpot client compiler. As mentioned earlier in Section 2.3, the HotSpot client compiler imposes a very small compilation delay, and yet generates code of only a slightly lower quality as compared to the much slower server compiler. Consequently, although the hot methods promoted to level-2 (server compiler) compilation may face delays in the compiler queue, its performance impact is much reduced since the program can still execute in level-1 (client compiler) optimized code. The small improvement in program performance with more compiler threads (up to seven) is again the result of reduction in the server compiler queue delay. However, overall we found that the compiler queue delay is much less of a factor with the tiered compilation policy.



(a) (b)

Figure 2.9: Effect of multiple compiler threads on multi-core compilation activity with the Server and Tiered VM

Our results also show large and progressively more severe performance losses with increasing compilation threshold aggressiveness as we increase the number of compiler threads past the number of (free) available processing cores. In order to explain these performance losses, we extend our framework to report additional measurements of compilation activity. Figures 2.9(a) and 2.9(b) respectively show compilation thread times and the number of methods compiled with the server and tiered VMs with their respective default compile thresholds and with increasing numbers of compiler threads, averaged over all the benchmarks. We find that, due to its extremely fast client compiler, the tiered VM employs a much more aggressive compile threshold, which enables it to compile more methods more quickly, and often finish the benchmark run faster, than the server VM. However, with its multi-level compilation strategy, this often results in a situation with many methods remaining in the level-2 (server) compilation queues at the end of the program run. Increasing the number of compilation threads enables more of these methods to be (re-)compiled during the application run. This additional compilation (with lower program speed benefit returns) obstruct application progress as the number of threads is raised beyond the limits of available hardware. Therefore, we conclude that *the number of compiler threads in the tiered VM should be set within the limits of available hardware in order to prevent sharp performance losses.*

Figure 2.8(b) presents the ratio of average program performance delivered by the VM with varying tiered compilation thresholds and compiler threads when compared to a single *baseline* performance (client compiler threshold of 2,000) with two (one client and one server) compiler

threads. Thus, employing a small number of compilation threads (< 10) typically achieves the best performance. This performance is much closer to optimal than the baseline performance with the server VM, although the server VM shows greater improvements as the number of compiler threads is increased.

Other trends in the graph in Figure 2.8(b) are similar to those presented with Figure 2.8(a) with the additional recommendation against employing overly conservative compiler policies on multi-core machines. Although conservative policies do a good job of reducing compilation overhead for single-core machines (Figure 2.6(b)), they can lower performance for multi-core machines due to a combination of two factors: (a) not compiling all the important program methods, and (b) causing a large delay in compiling the important methods. However, we also find that a wide range of compiler policies (from the client compiler thresholds of 4,000 to 20) achieve almost identical performance as long as compilation is free. This observation indicates that (a) not compiling the important methods (rather than the compiler queue delay) seems to be the dominant factor that can limit performance with the tiered compiler, and (b) compiling the less important program methods does not substantially benefit performance.

The plots in Figures 2.8(c) and 2.8(d) only employ the results of benchmarks that show very low or very high compilation activity respectively, and are constructed similar to the graph in Figure 2.8(b) in all other aspects. These graphs reiterate the earlier observation that program characteristics greatly influence its performance at different compiler aggressiveness. For benchmarks with only a few very hot methods (Figure 2.8(c)), varying compiler thresholds has little to no effect on overall performance. And Figure 2.8(d) shows that the average trends noticed across all of our benchmarks in Figure 2.8(b) are exaggerated when considered only over the programs displaying high compilation activity. This graph again indicates that the delay in compiling the hot methods does not seem to be a major factor affecting program run-time when using tiered compilation with a fast and good Level-1 compiler.

2.6 JIT Compilation on Many-Core Machines

Our observations regarding aggressive JIT compilation policies on modern multi-core machines in the last section were limited by our existing 8-core processor based hardware. In future years, architects and chip developers are expecting and planning a continuously increasing number of cores in modern microprocessors. It is possible that our conclusions regarding JIT compilation policies may change with the availability of more abundant hardware resources. However, processors with a large number of cores (or *many-cores*) are not easily available just yet. Therefore, in this section, we construct a unique experimental configuration to conduct experiments that investigate JIT compilation strategies for such future many-core machines.

Our experimental setup *estimates* many-core VM behavior using a single processor/core. To construct this setup, we first update our HotSpot VM to report the *category* of each operating system thread that it creates (such as, application, compiler, garbage-collector, etc.), and to also report the creation or deletion of any VM/program thread at runtime. Next, we modify the *harness* of all our benchmark suites to not only report the overall program execution time, but to also provide a break-down of the time consumed by each individual VM thread. We use the `/proc` file-system interface provided by the Linux operating system to obtain individual thread times, and employ the JNI interface to access this platform-specific OS feature from within a Java program. Finally, we also use the thread-processor-*affinity* interface methods provided by the Linux OS to enable our VM to choose the set of processor cores that are eligible to run each VM thread. Thus, on each new thread creation, the VM is now able to assign the processor affinity of the new VM thread (based on its category) to the set of processors specified by the user on the command-line. We use this facility to constrain all application and compiler threads in a VM to run on a single processor core.

Our experimental setup to evaluate the behavior of many-core (unlimited cores) application execution on a single-core machine is illustrated in Figure 2.10. Figure 2.10(a) shows a snapshot of one possible VM execution order with multiple compiler threads, with each thread running on a distinct core of a many-core machine. Our experimental setup employs the OS thread affinity

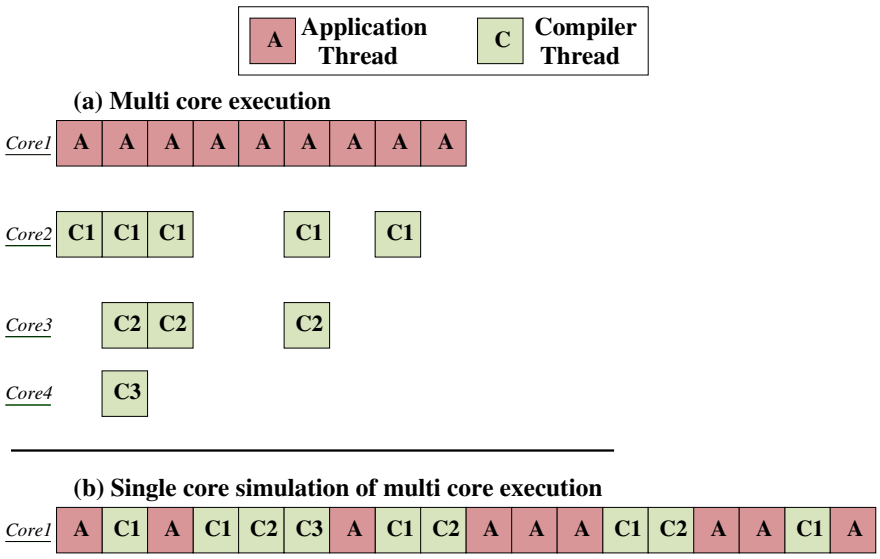


Figure 2.10: Simulation of multi-core VM execution on single-core processor

interface to force all application and compiler threads to run on a single core, and relies on the OS round-robin thread scheduling to achieve a corresponding thread execution order that is shown in Figure 2.10(b). It is important to note that JIT compilations in our simulation of many-core VM execution (on single-core machine) occur at about the same time relative to the application thread as on a physical many-core machine. Now, on a many-core machine, where each compiler thread runs on its own distinct core concurrently with the application thread, the total program run-time is equal to the application thread run-time alone, as understood from Figure 2.10(a). Therefore, our ability to precisely measure individual application thread times in our single-core simulation enables us to realistically emulate an environment where each thread has access to its own core. Note that, while this configuration is not by itself new, its application to measure "many-core" performance is novel. This framework, for the first time, allows us to study the behavior of different JIT compilation strategies with any number of compiler threads running on separate cores on future many-core hardware.

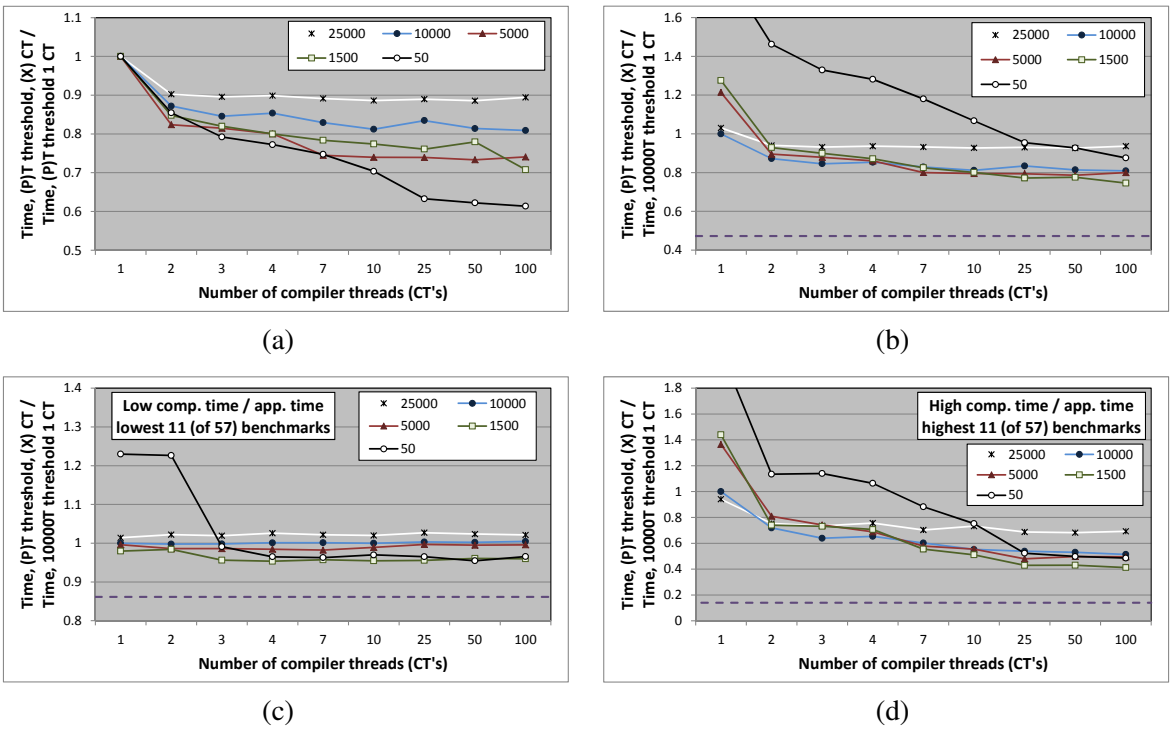
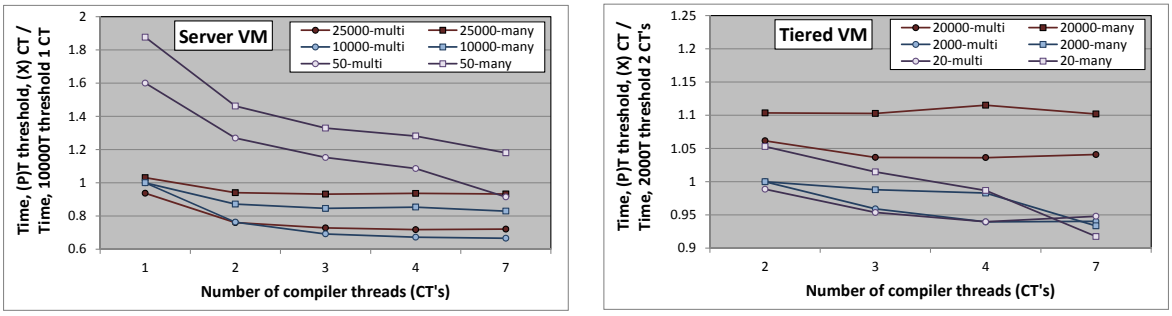


Figure 2.11: Effect of multiple compiler threads on many-core application performance with the HotSpot Server VM

2.6.1 Many-Core Compilation Policy with the HotSpot Server Compiler

We now employ our many-core experimental setup to conduct similar experiments to those done in Section 2.5.1. Figure 2.11 shows the results of these experiments and plots the average application thread times with varying number of compiler threads and compiler aggressiveness for all of our benchmark applications. These plots correspond with the graphs illustrated in Figure 2.7. In order to assess the accuracy of our simulation, we plot Figure 2.12(a), which shows a side-by-side comparison of a subset of the results for the multi and many-core configurations with 1-7 compiler threads. From these plots, we can see that the trends in these results are mostly consistent with our observations from the last section for a small (≤ 7) number of compiler threads. This similarity validates the ability of our simple simulation model to estimate the effect of JIT compilation policies on many-core machines, in spite of the potential differences between inter-core communication, cache models and other low-level microarchitectural effects.

Figure 2.11(a) shows that, unlike the multi-core plots in Figure 2.7(a), given unlimited comput-



(a) (b)

Figure 2.12: Comparison of multi- and many-core performance results for the server and tiered VM.

ing resources, application thread performance for aggressive compiler thresholds continues gaining improvements beyond a small number of compiler threads. Thus, the performance degradation for the more aggressive thresholds beyond about 7-10 compiler threads in the last section is, in fact, caused due to the limitations of the underlying 8-core hardware. This result shows the utility of our novel setup to investigate VM properties for future many-core machines. From the results plotted in Figure 2.11(b), we observe that the more aggressive compilation policies eventually (with $>$ 10 compiler threads) yield performance gains over the baseline server compiler threshold of 10,000 with one compiler thread. Additionally, we note that the difference between the baseline configuration and the optimal performance (indicated by the dashed line in Figure 2.11(b)) with our many-core simulation is similar to our results with multi-core machines.

We also find that isolating and plotting the performance of benchmarks with relatively small or relatively large amounts of compilation activity in our many-core configuration shows different trends than our complete set of benchmarks. As shown in Figure 2.11(c), increasing the number of compiler threads for benchmarks that spend relatively little time compiling does not have a significant impact on performance at any threshold. At the same time, early compilation of the (small number of) hot methods reduces the benchmark run-times at aggressive compilation thresholds. Alternatively, as seen from Figure 2.11(d), benchmarks with more compilation activity tend to show even starker performance improvements with increasing number of compiler threads. This result makes intuitive sense, as applications that require more compilation yield better performance

when we allocate additional compilation resources. Comparing the optimal performance over each set of benchmarks, we find that our many-core experiments show trends that are similar to our previous results – benchmarks with relatively little compilation activity achieve performance that is much closer to optimal, while benchmarks with relatively high compilation activity have more room for performance improvement.

2.6.2 Many-Core Compilation Policy with the HotSpot Tiered Compiler

In this section we analyze the results of experiments that employ our many-core framework to estimate the average run-time behavior of programs for the tiered compiler strategy with the availability of unlimited free compilation resources. The results in this section enable us to extend our observations from Section 2.5.2 to many-core machines. The results in Figures 2.13(a) and 2.13(b) again reveal that, in a wide range of compilation thresholds, changing compiler aggressiveness has a smaller effect on performance as compared to the single-level server compiler. As we expect, a side-by-side comparison of the multi and many-core results in Figure 2.12(b) shows that the trends in these results are mostly consistent with the results in Section 2.5.2 for a small number of compiler threads. The most distinctive observation that can be made from our many-core experiments is that, given sufficient computing resources, there is no significant performance loss with larger numbers of compiler threads since all compilation activity is considered free. Unfortunately, as noticed with the multi-core results, increasing the number of compiler threads is likely to only produce a modest impact on program run-time with the tiered compiler. This observation again indicates that techniques to reduce the delay in compiling hot methods are not as effective to improve run-time performance with the tiered compiler.

Figures 2.13(c) and 2.13(d) plot the same program run-time ratio as Figure 2.13(b), but only for benchmarks that have a very low or very high compilation activity. As observed in all similar plots earlier, benchmarks with low compilation activity have only a small number of very active methods, and all compiler aggressiveness levels produce similar performances. Benchmarks with a high compilation load mostly exaggerate the trends noticed over all benchmarks (Figure 2.13(b)).

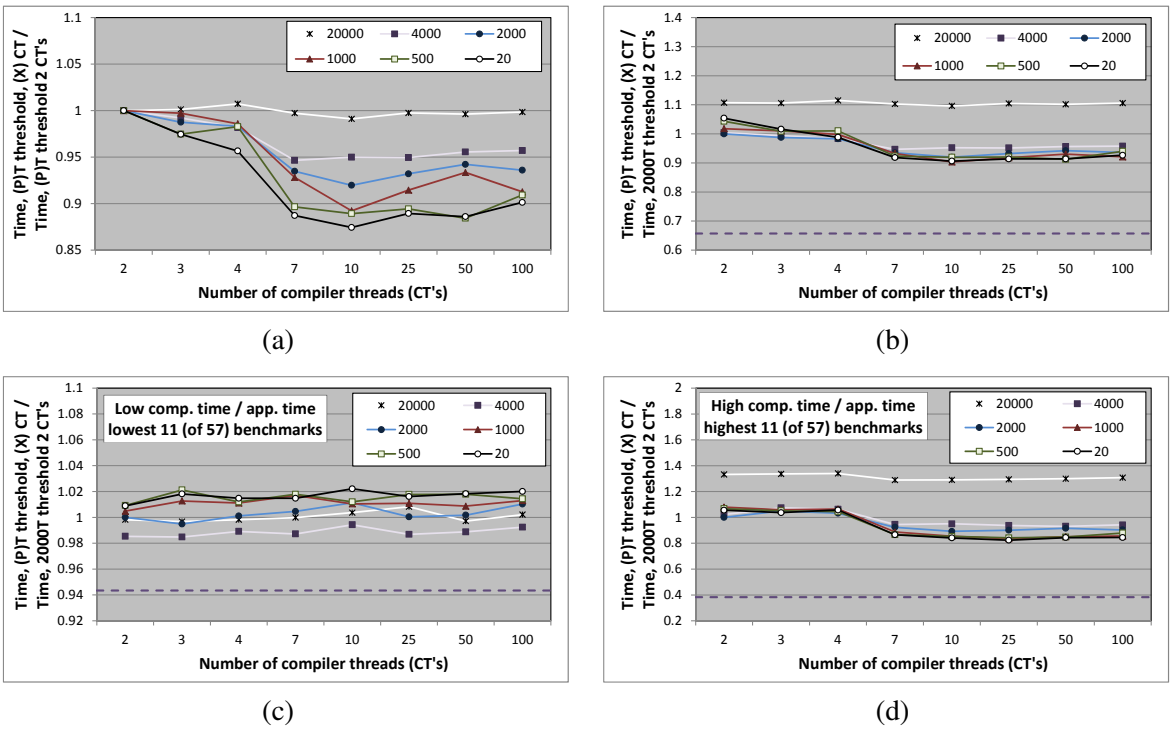


Figure 2.13: Effect of multiple compiler threads on many-core application performance with the HotSpot Tiered VM

Very conservative compiler thresholds cause large performance losses for such benchmarks. Additionally, with free compiler resources, higher compiler aggressiveness can produce marginally better results than the default threshold for such benchmarks by compiling and optimizing a larger portion of the program.

2.7 Effect of Priority-Based Compiler Queues

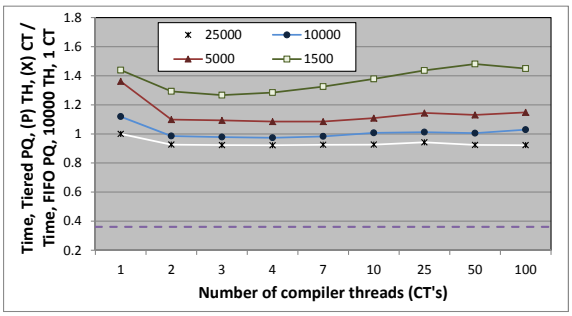
Aggressive compilation policies can send a lot of methods to compile, which may back-up the compile queue. Poor method ordering in the compiler queue may result in further delaying the compilation of the most important methods, as the VM spends its time compiling the less critical program methods. Delaying the generation of optimized code for the hottest methods will likely degrade application performance. An algorithm to effectively prioritize methods compiles may be able to nullify the harmful effects of back-up in the compiler queue. In this section, we study the effect of different compiler queue prioritization schemes on the server and tiered compiler

configurations.

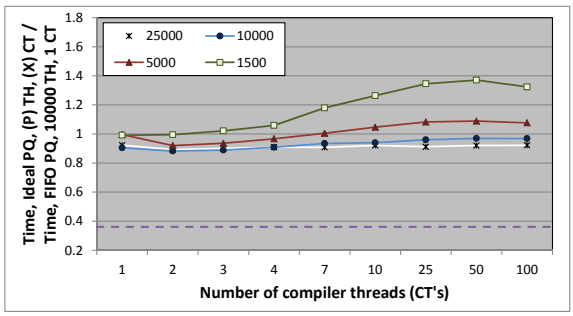
We present results of experiments with three different priority queue implementations. The first-in-first-out (FIFO) queue implementation is the default strategy employed by the HotSpot server compiler that compiles all methods in the order they are sent for compilation by the application threads. By default, the HotSpot *tiered* compiler uses a heuristic priority queue technique for ordering method compiles. When selecting a method for compilation with this heuristic, the tiered compiler computes an *event rate* for every eligible method and selects the method with the maximum event rate. The event rate is simply the sum of invocation and backedge counts per millisecond since the last dequeue operation. We modified the HotSpot VM to make the *FIFO* and *tiered* queue implementations available to the multi-stage tiered and single-stage server compilers respectively.

Both the FIFO and tiered techniques for ordering method compiles use a completely online strategy that only uses past program behavior to detect hot methods to compile to speed-up the remaining program run. Additionally, the more aggressive JIT compilation policies make their hotness decisions earlier, giving any online strategy an even reduced opportunity to accurately assess method priorities. Therefore, to set a suitable goal for the online compiler priority queue implementations, we attempt to construct an *ideal* strategy for ordering method compilations. An ideal compilation strategy should be able to precisely determine the actual hotness level of all methods sent to compile, and always compile them in that order. Unfortunately, such an ideal strategy requires knowledge of future program behavior.

In lieu of future program information, we devise a compilation strategy that prioritizes method compiles based on their total execution counts over an earlier profile run. With this strategy, the compiler thread always selects and compiles the method with the highest profiled execution counts from the available candidates in the compiler queue. Thus, our ideal priority-queue strategy requires a profile-run of every benchmark to determine its method hotness counts. We collect these total method hotness counts during this previous program run, and make them available to the ideal priority algorithm in the measured run. We do note that even our ideal profile-driven strategy may



(a)



(b)

Figure 2.14: Performance of the tiered and ideal compiler priority algorithms over FIFO for HotSpot server compiler on single-core machines

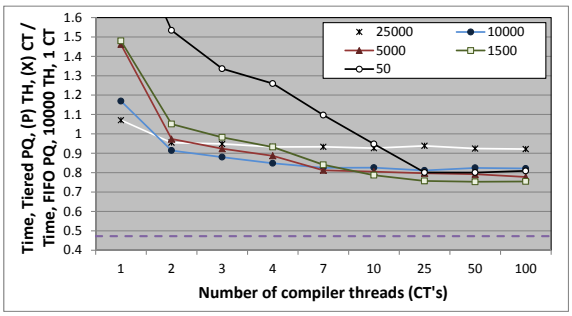
not achieve the *actual* best results because the candidate method with the highest hotness level may still not be the best method to compile at *that* point during program execution.

2.7.1 Priority-Based Compiler Queues in the HotSpot Server Compiler

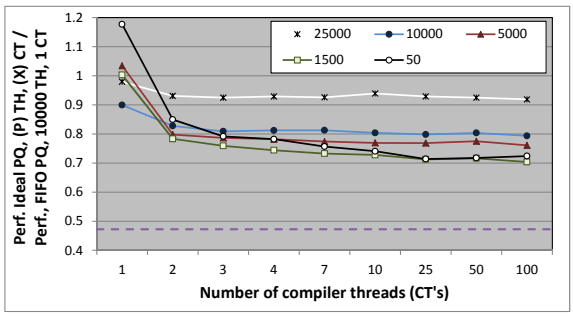
Our results in the earlier sections suggest that the relatively poor performance achieved by aggressive JIT compilation policies in the server compiler may be an artifact of the FIFO compiler queue that cannot adequately prioritize the compilations by *actual* hotness levels of application methods. Therefore, in this section, we explore and measure the potential of different priority queue implementations to improve the performance obtained by different JIT compilation strategies.

2.7.1.1 Single-Core Machine Configuration

Figures 2.14(a) and 2.14(b) show the performance benefit of the tiered and ideal compiler priority queue implementations, respectively. Each line in these graphs is plotted relative to the default FIFO priority queue implementation with a single compiler thread at the default threshold of 10,000 on single-core machines. These graphs reveal that the online tiered prioritization technique is not able to improve performance over the simple FIFO technique and actually results in a performance loss for a few benchmarks. In contrast, the VM performance with the ideal prioritization scheme shows that accurate assignment of method priorities is important, and allows the smaller compile thresholds to also achieve relatively good average program performance for small number



(a)



(b)

Figure 2.15: Performance of the tiered and ideal compiler priority algorithms over FIFO for HotSpot server compiler on many-core machines

of compiler threads.

We had also discovered (and reported in Section 2.4.2) that initiating a greater number of compiler threads on single-core machines results in compiling methods that are otherwise left uncompiled (in the compiler queue) upon program termination with fewer compiler threads. The resulting increase in the compilation overhead is not sufficiently compensated by the improved application efficiency, resulting in a net overall performance loss. We find that this effect persists regardless of the method priority algorithm employed. We do see that accurately ordering the method compiles enables the VM with our ideal priority queue implementation to obtain better performance than the best achieved with the FIFO queue.

2.7.1.2 Many-Core Machine Configuration

Figures 2.15(a) and 2.15(b) compare the performance results of using our tiered and ideal compiler priority queue, respectively, with a baseline VM that uses the simple FIFO-based compiler queue implementation with the compile threshold of 10,000 for many-core machines. The results with the ideal priority queue implementation show that appropriately sorting method compiles significantly benefits program performance at all threshold levels. At the same time, the performance benefits are more prominent for aggressive compile thresholds. This behavior is logical since more aggressive thresholds are more likely to flood the queue with low-priority compiles that delay the compilation of the *hotter* methods with the FIFO queue.

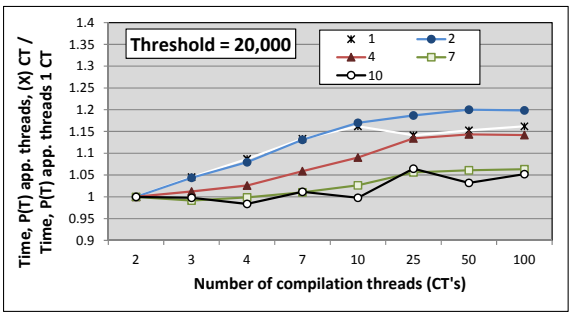
We also find that the best average benchmark performance with our ideal priority queue for every threshold plot is achieved with a smaller number of compiler threads, especially for the more aggressive compiler thresholds. This result shows that our ideal priority queue does realize its goal of compiling the hotter methods before the cold methods. The later lower priority method compilations seem to not make a major impact on program performance.

Finally, we can also conclude that using a good priority compiler queue allows more aggressive compilation policies (that compile a greater fraction of the program early) to improve performance over a less aggressive strategy on multi/many-core machines. Moreover, a small number of compiler threads is generally sufficient to achieve the best average application thread performance. Overall, the best aggressive compilation policy improves performance by about 30% over the baseline configuration, and by about 9% over the best performance achieved by the server VM's default compilation threshold of 10,000 with any number of compiler threads. Unfortunately, the online tiered prioritization heuristic is again not able to match the performance of the ideal priority queue. Thus, more research may be needed to devise better online priority algorithms to achieve the most effective overall program performance on modern machines.

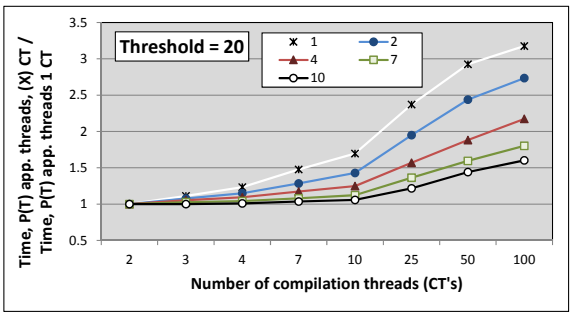
2.7.2 Priority-Based Compiler Queues in the HotSpot Tiered Compiler

As explained earlier, the HotSpot tiered configuration uses a simple and fast priority heuristic to order methods in the queue. In this section, we describe the impact of FIFO, tiered (default), and ideal prioritization algorithms for all the compiler queues in the tiered configuration.

We find that prioritizing method compiles has no significant effect on program performance at any compiler aggressiveness for the tiered compiler on all machines configurations. These results suggest that performance behavior with the tiered compilers is dominated by the very fast client compiler, which generates good quality code without causing a significant compiler queue backup. The very hot methods that are promoted to be further optimized by the server compiler do take time and may cause queue back-up. We find that larger server compiler overhead increases program runtime on single-core machines, but not on the many-core machines where compilation is free.



(a)



(b)

Figure 2.16: Effect of different numbers of application threads on single-core performance with the HotSpot Tiered VM

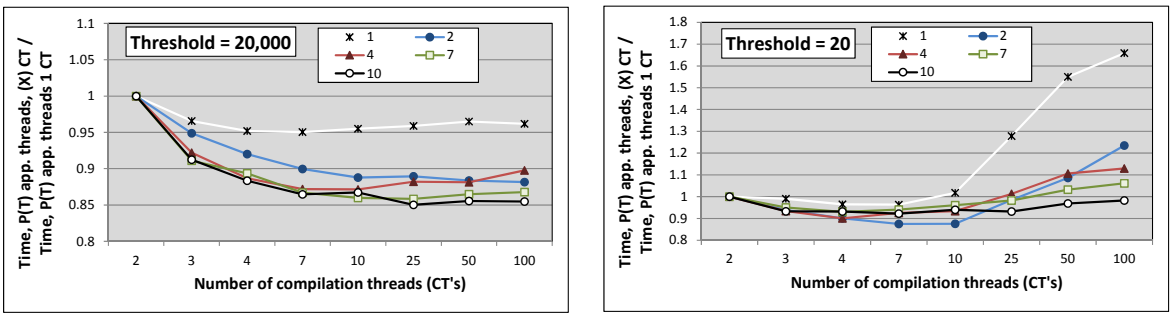
2.8 Effect of Multiple Application Threads

All our earlier experiments were primarily conducted with single-threaded benchmarks. However, real-world applications widely vary in the number and workload of concurrent application threads. In this section, we explore the effect of compiler aggressiveness and resources on the performance of benchmarks with different numbers of application threads.

Experiments in this section were conducted using 16 SPECjvm2008 benchmarks (all except *derby*).[5] Our other benchmark suites do not allow an easy mechanism to uniformly and precisely control the number of spawned application threads. While SPECjvm98 only permits runs with a fixed number of application threads, many of the DaCapo programs spawn a variable numbers of *internal* threads that cannot be controlled by the harness (see Table 2.2). For each VM configuration, we selected three compilation thresholds (least aggressive, most aggressive, and default) and ran the benchmarks with a different number of compiler and application threads to understand their interactions. Results with the server and tiered VMs show similar application–compiler thread interaction trends, and therefore, we only report results with the tiered VM in this section.

Figures 2.16(a) and 2.16(b) show the results of these experiments with the tiered VM on our single-core configuration for the least (CT=20,000) and most aggressive (CT=20) compile thresholds, respectively. Each line in the figures (plotted for a specific number of application threads

[5]SPECjvm2008 automatically scales the benchmark workload in proportion to the number of application threads. This causes the *derby* benchmark with a greater number of application threads to often fail with an out-of-memory error.



(a) (b)

Figure 2.17: Effect of different numbers of application threads on multi-core performance with the HotSpot Tiered VM

as indicated in the legend) shows the ratio of program speed with a different number of compiler threads to the performance with a single compiler thread *and that same number of application threads*. A separate baseline for each application thread configuration is necessitated because SPECjvm2008 employs a different workload size for each such setting. As explained previously in Section 2.4.2.2 (Figure 2.6), Figure 2.16 again shows that a fast and high-quality level-1 compiler that doesn't cause much queue delay is responsible for the loss in program performance with one (or few) application threads as the number of compiler threads are increased. However, we now observe that the magnitude of this loss diminishes as the number of application threads is increased. We believe this effect is caused by the change in relative application thread interference that is due to the compiler threads servicing less important program methods with limited hardware resources. In other words, the likelihood of the OS scheduler selecting an application thread is smaller when there are fewer application threads in the run queue. Thus, adding application threads diminishes the relative loss in performance by increasing the likelihood that an application thread runs.

Another interesting trend we noticed in both the server and tiered VMs can be see from the multi-core experiments displayed in Figure 2.17. We find that (especially at less aggressive compile thresholds) configurations with larger numbers of application threads tend to show slightly larger improvements as the number of compiler threads is increased. In programs with multiple application threads, the output of a compilation may be used simultaneously in threads execut-

ing in parallel. Thus, early compilations may benefit more of the application's computation in comparison to single-threaded applications.

We found that our many-core experiments also demonstrate very similar trends. However, in the absence of interference to the application threads with unlimited computation resources, we do not find any significant performance losses with larger numbers of compiler threads. These results also support the observation that configurations with more application threads tend to show more pronounced performance improvements from early compilation.

2.9 Conclusions

Many virtual machines now allow the concurrent execution of multiple compiler threads to exploit the abundant computing resources available in modern processors to improve overall program performance. It is expected that more aggressive JIT compilation strategies may be able to lower program run-time by compiling and optimizing more program methods early. The goal of this work is to explore the potential performance benefit of more aggressive JIT compilation policies for modern multi/many-core machines and VMs that support multiple simultaneous compiler threads. We explore the properties of two VM compiler configurations: a single-level highly optimizing (but slow) *server* compiler, and a multi-level *tiered* compiler. The HotSpot tiered compiler uses a very fast (but lightly optimizing) *client* compiler at the first stage, and a powerful SSA-based (*server*) compiler for re-compiling the very hot methods. Due to its re-compilations, the tiered compiler induces a higher compilation load compared to the single-level server compiler. Our experiments vary the hotness thresholds to control compiler aggressiveness, and employ different number of concurrent compiler threads to exploit free computation resources. We also develop a novel experimental framework to evaluate our goal for future many-core processors.

Results from our experiments allow us to make several interesting observations:

1. Properties of the tiered compiler are largely influenced by the fast client compiler that is able to obtain most of the performance benefits of the slower server compiler at a small fraction

of the compilation cost.

2. Program features, in particular the ratio of hot program methods, impact their performance behavior at different compiler aggressiveness levels. We found that programs with a low compilation to application time ratio are not significantly affected by varying compiler aggressiveness levels or by spawning additional compiler threads.

3. On single-core machines, compilation can impede application progress. The best compilation policy for such machines seems to be an aggressiveness level that only sends as many methods to compile as can be completely serviced by the VM in 1-2 compiler threads for most benchmarks.

4. For machines with multiple cores, methods compiled by a moderately aggressive compile threshold are typically sufficient to obtain the best possible performance. Compiling any more methods quickly results in diminishing returns and very minor performance gains.

5. Reducing the compiler queue delay and compiling early is more important to program performance than compiling all program methods. Spawning more compiler threads (at a given compilation aggressiveness) is an effective technique to reduce the compiler queue delay.

6. On single-core and multi-core machines, an excessive amount of compiler activity and threads may impede program progress by denying the application threads time to run on the CPU. This effect is reduced as the ratio of application to compiler threads increase. We also saw evidence that benchmarks with more application threads tend to show slightly sharper improvements from early compilation.

7. The tiered VM, with its extremely fast client compiler, is able to compile methods more quickly with fewer compilation resources, and typically outperforms the server VM on single-core machines or when the number of compiler threads is kept at a minimum. Alternatively, when there are additional computing resources available, increasing the number of compiler threads can sharply improve performance with the server VM.

8. Effective prioritization of method compiles is important to find and compile the most important methods early, especially for the slow, powerful, highly-optimizing compilers. However, additional research is necessary to find good *online* prioritization algorithms.

9. Initiating more compiler threads than available computing resources typically hurts performance.

Based on these observations we make the following recommendations regarding a good compilation policy for modern machines: (a) JIT compilers should use an adaptive compiler aggressiveness based on availability of free computing resources. At the very least VMs should employ two (sets of) compiler thresholds, a conservative (large) threshold when running on single-core processors, and moderately aggressive (small) threshold on multi/many-core machines. (b) Spawn as many compiler threads as available free compute cores (and as constrained by the specific power budget). (c) Employ an effective priority queue implementation to reduce the compilation queue delay for the slower compilers. (d) The more complex tiered compilation strategies used in some of the existing state-of-the-art JVMs can achieve the most effective startup program performance with minimal compiler threads and little need for effective prioritization of method compiles. We believe that our comprehensive research will guide future VM developers in making informed decisions regarding how to design and implement the most effective JIT compilation policies to achieve the best application performance.

2.10 Future Work

This work presents several interesting avenues for future research. First, this work shows that the availability of abundant computation resources in future machines will allow the possibility of program performance improvement by early compilation of a greater fraction of the program. With the development of profile-driven optimization phases, future work will have to consider the effect of early compilation on the amount of collected profile information and resulting impact on generated code. Additionally, researchers may also need to explore the interaction of increased

compiler activity with garbage collection. More native code produced by aggressive JIT compilation can raise memory pressure and garbage collection overheads, which may then affect program non-determinism due to the increased pause times associated with garbage collections. Second, in this chapter we explored some priority queue implementations that may be more suitable for aggressive compilation policies, especially with a slow, highly optimizing compiler. We plan to continue our search for better method prioritization schemes in the future. Third, this work shows that the optimal settings for compilation threshold and the number of compiling threads depend on factors, such as application characteristics, that cannot be determined statically. Thus, we plan to conduct experiments to study the performance potential of adaptively scaling these parameters at runtime. Fourth, this work primarily focuses on exploring *if* and *when* to compile program methods to maximize overall program performance for modern machines. We do consider some aspects of *how* to compile with the tiered HotSpot configuration. However, how to compile program regions is a much broader research topic that includes issues such as better *selection* and *ordering* of optimizations at different compilation levels. The following chapter presents research we conducted to address some of these issues in the HotSpot VM. We plan to exploit the observations from this work to focus on optimizations (and methods) with the greatest impact on program performance and build new and more effective online models. Finally, we are currently also conducting similar experiments in other virtual machines (JikesRVM) to see if our conclusions from this work hold across different VMs. Later, we plan to also validate our results for different processor architectures.

Chapter 3

Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation

Phase selection is the process of customizing the applied set of compiler optimization phases for individual functions or programs to improve performance of generated code. Researchers have recently developed novel feature-vector based heuristic techniques to perform phase selection during online JIT compilation. While these heuristics improve program *startup* speed, *steady-state* performance was not seen to benefit over the default fixed single sequence baseline. Unfortunately, *it is still not conclusively known whether this lack of steady-state performance gain is due to a failure of existing online phase selection heuristics, or because there is, indeed, little or no speedup to be gained by phase selection in online JIT environments*. The goal of this work is to resolve this question, while examining the phase selection related behavior of optimizations, and assessing and improving the effectiveness of existing heuristic solutions.

We conduct experiments to find and understand the potency of the factors that can cause the phase selection problem in JIT compilers. Next, using long-running genetic algorithms we determine that program-wide and method-specific phase selection in the HotSpot JIT compiler can

produce *ideal* steady-state performance gains of up to 15% (4.3% average) and 44% (6.2% average) respectively. We also find that existing state-of-the-art heuristic solutions are unable to realize these performance gains (in our experimental setup), discuss possible causes, and show that exploiting knowledge of optimization phase behavior can help improve such heuristic solutions. Our work develops a robust open-source production-quality framework using the HotSpot JVM to further explore this problem in the future.

3.1 Introduction

An optimization phase in a compiler transforms the input program into a semantically equivalent version with the goal of improving the performance of generated code. Quality compilers implement many optimizations. Researchers have found that the set of optimizations producing the best quality code varies for each method/program and can result in substantial performance benefits over always applying any single optimization sequence (Hoste & Eeckhout, 2008; Cavazos & O'Boyle, 2006). *Optimization phase selection* is the process of automatically finding the best set of optimizations for each method/program to maximize performance of generated code, and is an important, fundamental, but unresolved problem in compiler optimization.

We distinguish phase selection from the related issue of *phase ordering*, which explores the effect of different orderings of optimization phases on program performance. We find that although phase ordering is possible in some research compilers, such as VPO (Kulkarni et al., 2007b) and Jikes Research Virtual Machine (VM) (Kulkarni & Cavazos, 2012), reordering optimization phases is extremely hard to support in most production systems, including GCC (Fursin et al., 2011) and the HotSpot VM, due to their use of multiple intermediate formats and complex inherent dependencies between optimizations. Therefore, our work for this chapter conducted using the HotSpot JVM only explores the problem of phase selection by selectively turning optimization phases ON and OFF.

The problem of optimization selection has been extensively investigated by the static compiler

community (Almagor et al., 2004; Hoste & Eeckhout, 2008; Fursin et al., 2011; Triantafyllis et al., 2003). Unfortunately, techniques employed to successfully address this problem in static compilers (such as *iterative compilation*) are not always applicable to dynamic or JIT (Just-in-Time) compilers. Since compilation occurs at runtime, one over-riding constraint for dynamic compilers is the need to be fast so as to minimize interference with program execution and to make the optimized code available for execution sooner. To realize fast online phase selection, researchers have developed novel techniques that employ feature-vector based machine-learning heuristics to quickly customize the set of optimizations applied to each method (Cavazos & O'Boyle, 2006; Sanchez et al., 2011). Such heuristic solutions perform the time-consuming task of *model learning* offline, and then use the models to quickly customize optimization sets for individual programs/methods online during JIT compilation.

It has been observed that while such existing online phase selection techniques improve program *startup* (application + compilation) speed, they do not benefit throughput or *steady-state* performance (program speed after all compilation activity is complete). Program throughput is very important to many longer-running applications. Additionally, with the near-universal availability of increasingly parallel computing resources in modern multi/many-core processors and the ability of modern VMs to spawn multiple asynchronous compiler threads, researchers expect the compilation overhead to become an even smaller component of the total program runtime (Kulkarni, 2011). Consequently, achieving steady-state or *code-quality* improvements is becoming increasingly important for modern systems. Unfortunately, while researchers are still investing in the development of new techniques to resolve phase selection, *we do not yet conclusively know whether the lack of steady-state performance gain provided by existing online phase selection techniques is a failure of these techniques, or because there is, indeed, little or no speedup to be gained by phase selection in online environments*. While addressing this primary question, we make the following contributions in this work.

1. We conduct a thorough analysis to understand the phase selection related behavior of JIT compiler optimization phases and identify the potency of factors that could produce the

phase selection problem,

2. We conduct long-running (genetic algorithm based) iterative searches to determine the *ideal* benefits of phase selection in online JIT environments,
3. We evaluate the accuracy and effectiveness of existing state-of-the-art online heuristic techniques in achieving the benefit delivered by (the more expensive) iterative search techniques and discuss improvements, and
4. We construct a robust open-source framework for dynamic JIT compiler phase selection exploration in the standard Sun/Oracle HotSpot Java virtual machine (JVM) (Paleczny et al., 2001). Our framework prepares 28 optimizations in the HotSpot compiler for individual fine-grain control by different phase selection algorithms.

The rest of the chapter is organized as follows. We present related work in the next section. We describe our HotSpot based experimental framework and benchmark set in Section 3.3. We explore the behavior of HotSpot compiler optimizations and present our observations in Section 3.4. We present our results on the ideal case performance benefits of phase sequence customization at the whole-program and per-method levels in Section 3.5. We determine the effectiveness of existing feature-vector based heuristic techniques and provide feedback on improving them in Section 3.6. Finally, we present our planned future work and the conclusions of this study in Sections 3.7 and 3.8 respectively.

3.2 Background and Related Work

In this section we provide some overview and describe related works in the area of optimization phase selection research. Phase selection and ordering are long-standing problems in compiler optimization. Several studies have also discovered that there is no single optimization set that can produce optimal code for every method/program (Almagor et al., 2004). A common technique to address the phase selection problem in static compilers is to iteratively evaluate the performance of many/all different phase sequences to find the best one for individual methods or programs.

Unfortunately, with the large number of optimizations present in modern compilers (say, n), the search space of all possible combinations of optimization settings (2^n) can be very large. Therefore, researchers have in the past employed various techniques to reduce the space of potential candidate solutions. Chow and Wu applied a technique called fractional factorial design to systematically design a series of experiments to select a good set of program-specific optimization phases by determining interactions between phases (Chow & Wu, 1999). Haneda et al. employed statistical analysis of the effect of compiler options to prune the phase selection search space and find a single compiler setting for a collection of programs that performs better than the standard settings used in GCC (Haneda et al., 2005b). Pan and Eigenmann developed three heuristic algorithms to quickly select good compiler optimization settings, and found that their combined approach that first identifies phases with negative performance effects and greedily eliminates them achieves the best result (Pan & Eigenmann, 2008). Also related is the work by Fursin et al. who develop a GCC-based framework (MILEPOST GCC) to automatically extract program *features* and learn the best optimizations across programs. Given a new program to compile, this framework can correlate the program's features with the closest program seen earlier to apply a customized and potentially more effective optimization combination (Fursin et al., 2011).

Machine-learning techniques, such as genetic algorithms and hill-climbing, have been commonly employed to search for the best set or ordering of optimizations for individual programs/methods. Cooper et al. were among the first to use machine-learning algorithms to quickly and effectively search the phase selection search space to find program-level optimization sequences to reduce code-size for embedded applications (Cooper et al., 1999; Almagor et al., 2004). Hoste and Eeckhout developed a system called *COLE* that uses genetic algorithm based multi-objective evolutionary search algorithm to automatically find pareto optimal optimization settings for GCC (Hoste & Eeckhout, 2008). Kulkarni et al. compared the proficiency of several machine-learning algorithms to find the best phase sequence as obtained by their exhaustive search strategy (Kulkarni et al., 2007b). They observed that search techniques such as genetic algorithms achieve benefit that is very close to that best performance. *We use this result in our current study*

to characterize the GA-delivered best performance as a good indicator of the performance limit of phase selection in dynamic compilers.

Also related is their more recent work that compares the ability of GA-based program and function-level searches to find the best phase sequence, and finds the finer-granularity of function-level searches to achieve better overall results in their static C compiler, VPO (Kulkarni et al., 2010). All these above studies were conducted for static compilers. Instead, in this work, we use the GA searches to determine the performance limits of phase selection in dynamic compilers.

While program-specific GA and other iterative searches may be acceptable for static compilers, they are too time-consuming for use in dynamic compilers. Consequently, researchers have developed novel techniques to quickly customize phase sequences to individual methods in dynamic JIT compilers. Cavazos and O'Boyle employed the technique of logistic regression to learn a predictive model offline that can later be used in online compilation environments (like their Jikes Research VM) to derive customized optimization sequences for methods based on its features (Cavazos & O'Boyle, 2006). Another related work by Sanchez et al. used support vector machines to learn and discover method-specific compilation sequences in IBM's (closed-source) production JVM (Sanchez et al., 2011). This work used a different learning algorithm and was conducted in a production JVM. While these techniques reduce program startup times due to savings in the compilation overhead, the feature-vector based online algorithm was not able to improve program steady-state performance over the default compiler configuration. Additionally, none of the existing works attempt to determine the potential benefit of phase selection to improve code-quality in a dynamic compiler. While resolving this important question, we also evaluate the success of existing heuristic schemes to achieve this best potential performance benefit in the HotSpot production JVM.

In this chapter, we also investigate the behavior of optimizations to better understand the scope and extent of the optimization selection problem in dynamic compilers. Earlier researchers have attempted to measure the benefit of dynamic optimizations, for example, to overcome the overheads induced by the safety and flexibility constraints of Java (Ishizaki et al., 1999), in cross-platform

Optimization Phase	Description
aggressive copy coalescing	Perform aggressive copy coalescing after coming out of SSA form (before register allocation).
block layout by frequency	Use edge frequencies to drive block ordering.
block layout rotate loops	Allow back branches to be fall through when determining block layout.
conditional constant prop.	Perform optimistic sparse conditional constant propagation until a fixed point is reached.
conservative copy coalescing	Perform conservative copy coalescing during register allocation (RA). Requires RA is enabled.
do escape analysis	Identify and optimize objects that are accessible only within one method or thread.
eliminate allocations	Use escape analysis to eliminate allocations.
global code motion	Hoist instructions to block with least execution frequency.
inline	Replace (non accessor/mutator) method calls with the body of the method.
inline accessors	Replace accessor/mutator method calls with the body of the method.
instruction scheduling	Perform instruction scheduling after register allocation.
iterative global value numbering (GVN)	Iteratively replaces nodes with their values if the value has been previously recorded (applied in several places after parsing).
loop peeling	Peel out the first iteration of a loop.
loop unswitching	Clone loops with an invariant test and insert a clone of the test that selects which version to execute.
optimize null checks	Detect implicit null check opportunities (e.g. null checks with suitable memory operations nearby use the memory operation to perform the null check).
parse-time GVN	Replaces nodes with their values if the value has been previously recorded during parsing.
partial loop peeling	Partially peel the top portion of a loop by cloning and placing one copy just before the new loop head and the other copy at the bottom of the new loop (also known as loop rotation).
peephole remove copies	Apply peephole copy removal immediately following register allocation.
range check elimination	Split loop iterations to eliminate range checks.
reassociate invariants	Re-associates expressions with loop invariants.
register allocation	Employ a Briggs-Chaitin style graph coloring register allocator to assign registers to live ranges.
remove useless nodes	Identify and remove useless nodes in the ideal graph after parsing.
split if blocks	Folds some branches by cloning compares and control flow through merge points.
use loop predicate	Generate a predicate to select fast/slow loop versions.
use super word	Transform scalar operations into packed (super word) operations.
eliminate auto box*	Eliminate extra nodes in the ideal graph due to autoboxing.
optimize fills*	Convert fill/copy loops into an intrinsic method.
optimize strings*	Optimize strings constructed by StringBuilder.

Table 3.1: Configurable optimizations in our modified HotSpot compiler. Optimizations marked with $*$ are disabled in the default compiler.

implementations (Ishizaki et al., 2003), and to explore optimization synergies (Lee et al., 2006). However, none of these works perform their studies in the context of understanding the optimization selection problem for JIT compilers.

3.3 Experimental Framework

In this section, we describe the compiler and benchmarks used, and the methodology employed for our experiments.

3.3.1 Compiler and Benchmarks

We perform our study using the server compiler in Sun/Oracle's HotSpot Java virtual machine (build 1.6.0_25-b06) (Paleczny et al., 2001). Similar to many production compilers, *HotSpot imposes a strict ordering on optimization phases* due to the use of multiple intermediate representations and documented or undocumented assumptions and dependencies between different phases. Additionally, the HotSpot compiler applies a fixed set of optimizations to every method it compiles. The compilation process parses the method's bytecodes into a static single assignment (SSA) representation known as the *ideal graph*. Several optimizations, including method inlining, are applied as the method's bytecodes are parsed. The compiler then performs a fixed set of optimizations on the resultant structure before converting to machine instructions, performing scheduling and register allocation, and finally generating machine code.

The HotSpot JVM provides command-line flags to optionally enable or disable several optimizations. However, many optimization phases do not have such flags, and some also generate/update analysis information used by later stages. We modified the HotSpot compiler to provide command-line flags for most optimization phases, and factored out the analysis calculation so that it is computed regardless of the optimization setting. Some transformations, such as *constant folding* and *instruction selection*, are required by later stages to produce correct code and are hard to effectively disable. We also do not include a flag for *dead code elimination*, which is performed continuously by the structure of the intermediate representation and would require much more invasive changes to disable. We perform innovative modifications to deactivate the compulsory phases of *register allocation* (RA) and *global code motion* (GCM). The disabled version of RA assumes every register conflicts with every live range, and thus, always spills live ranges to memory. Likewise, the disabled version of GCM schedules all instructions to execute as late as possible and does not attempt to hoist instructions to blocks that may be executed much less frequently. Finally, we modified the HotSpot JVM to accept binary sequences describing the application status (ON/OFF) of each phase at the program or method-level. Thus, *while not altering any optimization algorithm or the baseline compiler configuration, we made several major updates to the HotSpot*

JIT compiler to facilitate its use during phase selection research. Table 3.1 shows the complete set of optimization phases that we are now able to optionally enable or disable for our experiments.

Our experiments were conducted over applications from two suites of benchmarks. We use all SPECjvm98 benchmarks (SPEC98, 1998) with two input sizes (10 and 100), and 12 (of 14) applications from the DaCapo benchmark suite (Blackburn et al., 2006) with the small and default inputs. Two DaCapo benchmarks, *tradebeans* and *tradesoap*, are excluded from our study since they do not always run correctly with our *default* HotSpot VM.

3.3.2 Performance Measurement

One of the goals of this research is to quantify the performance benefit of optimization phase selection *with regards to generated code quality* in a production-grade dynamic compiler. Therefore, the experiments in this study discount compilation time and measure the *steady-state* performance. In the default mode, the VM employs *selective compilation* and only compiles methods with execution counts that exceed the selected threshold. Our experimental setup first determines this set of (hot) methods compiled during the *startup* mode for each benchmark. All our steady-state experiments only compile this hot method set for all its program runs. Both the SPECjvm98 and DaCapo harness allow each benchmark to be iterated multiple times in the same VM run. During our steady-state program runs we disable background compilation to force all these hot methods to be compiled in the first program iteration. We modify our VM to reset execution counts after each iteration to prevent methods from becoming hot (and getting compiled) in later iterations. We allow each benchmark to iterate five more times and record the median runtime of these iterations as the steady-state program run-time. To account for inherent timing variations during the benchmark runs, *all the performance results in this chapter report the average and 95% confidence intervals over* ten *steady-state runs* using the setup described by Georges et al. (Georges et al., 2007).

All experiments were performed on a cluster of Dell PowerEdge 1850 server machines running Red Hat Enterprise Linux 5.1 as the operating system. Each machine has four 64-bit 2.8GHz Intel Xeon processors, 6GB of DDR2 SDRAM, and a 4MB L2 cache. Our HotSpot VM uses

the stand-alone server compiler and the default garbage collector settings for "server-class" machines (http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper 150215.pdf, 2006) ("parallel collector" GC, initial heap size is 96MB, maximum is 1GB). We make no attempt to restrict or control GC during our experiments. Finally, there are no hyperthreading or frequency scaling techniques of any kind enabled during our experiments.

3.4 Analyzing Behavior of Compiler Optimizations for Phase Selection

Compiler optimizations are designed to improve program performance. Therefore, it is often (naïvely) expected that always applying (turning ON) all available optimizations to all program regions should generate the best quality code. However, optimizations operating on the same program code and competing for finite machine resources (registers) may interact with each other. Such interactions may remove opportunities for later optimizations to generate even better code. Additionally, program performance is often very hard for the compiler to predict on the current generation of machines with complex architectural and micro-architectural features. Consequently, program transformations performed by an optimization may not always benefit program execution speed. The goal of effective phase selection is to find and disable optimizations with negative effects for each program region. In this section we conduct a series of experiments to explore important optimization selection issues, such as why and when is optimization selection effective for standard dynamic JIT compilers. We are also interested in finding indicators to suggest that customizing optimization selections for individual programs or methods is likely to benefit performance. We report several interesting observations that help explain both the prior as well as our current results in phase selection research for dynamic JIT compilers.

3.4.1 Experimental Setup

Our setup to analyze the behavior of optimization phases is inspired by Lee et al.'s framework to determine the benefits and costs of compiler optimizations (Lee et al., 2006). Our experimental configuration (*defOpt*) uses the default HotSpot server compilation sequence as baseline. The execution time of each benchmark with this baseline ($T(OPT < defOpt >)$) is compared with its time obtained by a JIT compiler that disables one optimization (x) at a time ($T(OPT < defOpt - x >)$). We use the following fraction to quantify the effect of HotSpot optimizations in this configuration.

$$\frac{T(OPT < defOpt - x >) - T(OPT < defOpt >)}{T(OPT < defOpt >)} \tag{3.1}$$

Each experimental run disables only one optimization (out of 25) from the optimization set used in the default HotSpot compiler. Equation 3.1 computes a negative value if removing the corresponding optimization, x, from the baseline optimization set improves performance (reduces program runtime) of the generated code. In other words, a negative value implies that including that optimization harms performance of the generated code. The HotSpot JIT compiler uses an individual method for its compilation unit. Therefore, in this section we evaluate the effect of compiler optimizations over distinct program methods.

Our experiments in this section are conducted over 53 *hot* focus methods over all the programs in our benchmark suite. These focus methods are selected because each comprises *at least* 10% of its respective benchmark run-time. More details on the rationale and selection of focus methods, as well as a complete list of these methods, are provided in Section 3.5.3.1.

3.4.2 Results and Observations

Figure 3.1 (left Y-axis, bar-plot) illustrates the *accumulated* negative and positive impact of each optimization calculated using Equation 3.1 over all our 53 individual program methods. For each HotSpot optimization, Figure 3.1 (right Y-axis, line-plot) also shows the number of program meth-

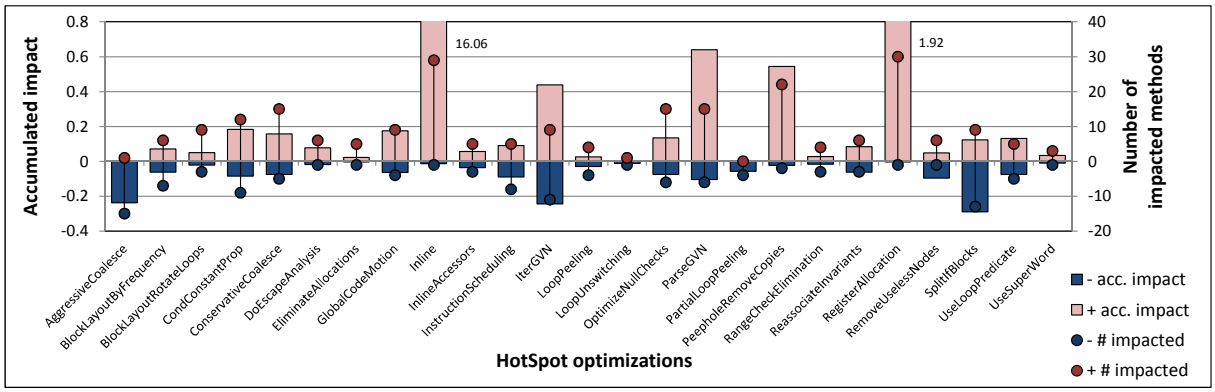


Figure 3.1: **Left Y-axis:** Accumulated positive and negative impact of each HotSpot optimization over our focus methods (non-scaled). **Right Y-axis:** Number of focus methods that are positively or negatively impacted by each HotSpot optimization.

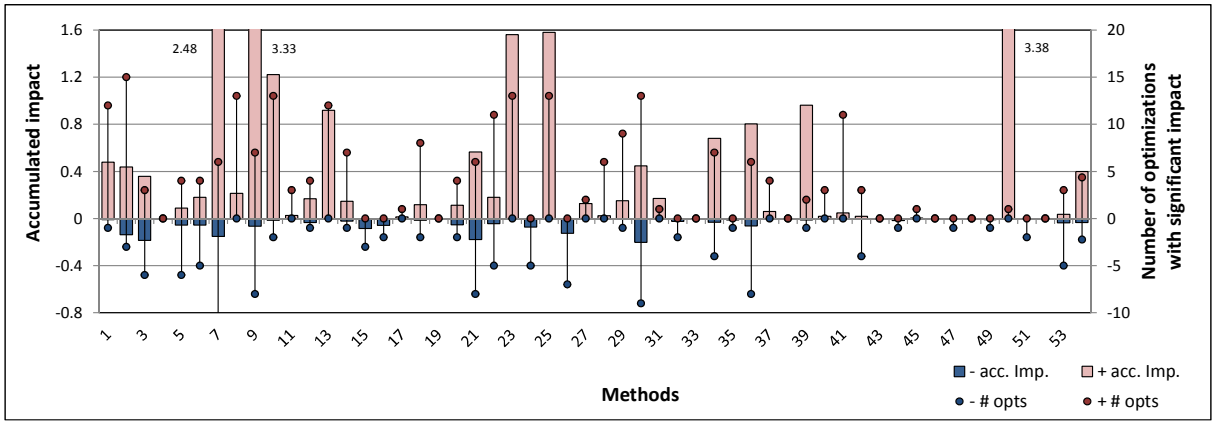


Figure 3.2: **Left Y-axis:** Accumulated positive and negative impact of the 25 HotSpot optimizations for each focus method (non-scaled). **Right Y-axis:** Number of optimizations that positively or negatively impact each focus method.

ods that witness a negative or positive impact. These results enable us to make several important observations regarding the behavior of optimizations in the HotSpot JIT compiler. **First**, the results validate the claims that optimizations are not always beneficial to program performance. This observation provides the motivation and justification for further developing and exploring effective phase selection algorithms to enable the JIT compiler to generate the best possible output code for each method/program. **Second**, we observe that *most* optimizations in the HotSpot JIT compiler produce, at least occasional, negative effects. This observation indicates that eliminating the optimization phase selection issue may require researchers to understand and update several different compiler optimizations, which makes a compiler design-time solution very hard. **Third**, most op-

timizations do not negatively impact a large number of program methods, and the typical negative impact is also not very high. However, we also find optimizations, including *AggressiveCoalesce*, *IterGVN*, and *SplitIfBlocks*, that, rather than improving, show a degrading performance impact more often. This result is surprising since dynamic compilers generally only provide the more conservative compiler optimizations.[1] Thus, this study finds optimizations that need to be urgently analyzed to alleviate the optimization selection problem in HotSpot. **Fourth**, we unexpectedly find that most of the optimizations in the HotSpot JIT compiler only have a marginal individual influence on performance. We observe that *method inlining* is by far the most beneficial compiler optimization in HotSpot, followed by *register allocation*.[2]

Figure 3.2 plots the accumulated positive and negative optimization impact (on left Y-axis, bar-plot) and the number of optimizations that impact performance (on right Y-axis, line-plot) for each of our focus methods represented along the X-axis. These results allow us to make two other observations that are particularly enlightening. **First**, there are typically not many optimizations that degrade performance for any single method (2.2 out of 25, on average). More importantly, even for methods with several individual degrading optimizations, the accumulated negative impact is never very high. This result, in a way, tempers the expectations of performance improvements from ideal phase selection in JIT compilers (particularly, HotSpot). In other words, we can only expect customized phase selections to provide modest performance benefits for individual methods/programs in most cases. **Second**, for most methods, there are only a few optimizations (4.36 out of 25, on average) that benefit performance. Thus, there is a huge potential for saving compilation overhead during program *startup* by disabling the *inactive* optimizations. It is this, hitherto unreported, attribute of JIT compilers that enables the online feature-vector based phase selection

[1]Compare the 28 optimization flags in HotSpot with over 100 such flags provided by GCC.

[2]Method inlining is a difficult optimization to control. Our experimental setup, which uses a fixed list of methods to compile, may slightly exaggerate the performance impact of disabling method inlining because some methods that would normally be inlined may not be compiled at all if they are not in the hot method list. To avoid such exaggeration, one possibility is to detect and compile such methods when inlining is disabled. However, an inlined method (say, P) that is not otherwise compiled spends its X inlined invocations in compiled code, but other Y invocations in interpreted code. With inlining disabled for the focus method, if P is compiled then it will spend all 'X+Y' invocations in compiled code. We chose the exaggeration because we found that it was very uncommon for methods not in the fixed list to still be inlinable.

algorithms to improve program startup performance in earlier works (Cavazos & O'Boyle, 2006; Sanchez et al., 2011).

Finally, we note that although this simple study provides useful information regarding optimization behavior, it may not capture all possible optimization interactions that can be simultaneously active in a single optimization setting for a method. For example, phase interactions may cause compiler optimization phases that degrade performance when applied alone to improve performance when combined with other optimizations. However, these simple experiments provided us with both the motivation to further explore the potential of phase selection for dynamic compilers, while lowering our expectations for large performance benefits.

3.5 Limits of Optimization Selection

Most dynamic JIT compilers apply the same set of optimization phases to all methods and programs. Our results in the last section indicate the potential for performance gains by customizing optimization phase selection for individual (smaller) code regions. In this section we conduct experiments to quantify the steady-state speed benefits of customizing optimization sets for individual programs/methods in JIT compilers. The large number of optimizations in HotSpot makes it unfeasible to perform *exhaustive* optimization selection search space evaluation. Earlier research has demonstrated that genetic algorithms (GA) are highly effective at finding near-optimal phase sequences (Kulkarni et al., 2007b). Therefore, we use a variant of a popular GA to find effective program-level and method-level optimization phase selection solutions (Cooper et al., 1999). Correspondingly, *we term the benefit in program run-time achieved by the GA derived phase sequence over the default HotSpot VM as the ideal performance benefit of phase selection for each program/method*.

We also emphasize that it is impractical to employ a GA-based solution to customize optimization sets in an online JIT compilation environment. Our program-wide and method-specific GA experiments are intended to only determine the performance limits of phase selection. We use

these limits in the next section to evaluate the effectiveness of existing state-of-the-art heuristics to specialize optimization sets in online JIT compilers.

3.5.1 Genetic Algorithm Description

In this section we describe the genetic algorithm we employ for our phase selection experiments. Genetic algorithms are heuristic search techniques that mimic the process of natural evolution (Goldberg, 1989). *Genes* in the GA correspond to binary digits indicating the ON/OFF status of an optimization. *Chromosomes* correspond to optimization phase selections. The set of chromosomes currently under consideration constitutes a *population*. The evolution of a population occurs in *generations*. Each generation evaluates the *fitness* of every chromosome in the current population, and then uses the operations of *crossover* and *mutation* to create the next population. The number of *generations* specifies the number of population sets to evaluate. Chromosomes in the first GA generation are randomly initialized. After evaluating the performance of code generated by each chromosome in the *population*, they are sorted in decreasing order of performance. During *crossover*, 20% of chromosomes from the poorly performing half of the population are replaced by repeatedly selecting two chromosomes from the better half of the population and replacing the lower half of each chromosome with the lower half of the other to produce two new chromosomes each time. During *mutation* we flip the ON/OFF status of each gene with a small probability of 5% for chromosomes in the upper half of the population and 10% for the chromosomes in the lower half. The chromosomes replaced during crossover, as well as (up to five) chromosome(s) with performance(s) within one standard deviation of the best performance in the generation are not mutated. The *fitness criteria* used by our GA is the steady-state performance of the benchmark. For this study, we have 20 chromosomes in each population and run the GA for 100 generations. We have verified that 100 generations are sufficient for the GA to reach saturation in most cases. To speed-up the GA runs, we developed a parallel GA implementation that can simultaneously evaluate multiple chromosomes in a generation on a cluster of identically-configured machines.

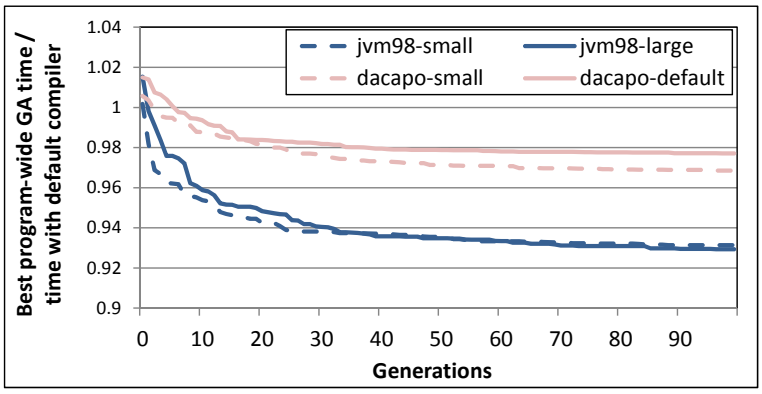


Figure 3.3: Average performance of best GA sequence in each generation compared to the default compiler.

3.5.2 Program-Wide GA Results

In this experiment we use our GA to find unique optimization selections for each of our benchmark-input pairs. Figure 3.3 plots the performance of code compiled with the best optimization set found by the GA in each generation as compared to the code generated by the default compiler sequence averaged over all programs for each benchmark suite. This figure shows that most (over 75%) of the average performance gains are realized in the first few (20) GA generations. Also, over 90% of the best average performance is obtained after 50 generations. Thus, 100 generations seem sufficient for our GA to converge on its near-best solution for most benchmarks. We also find that the SPECjvm98 benchmarks benefit more from optimization specialization than the DaCapo benchmarks. As expected, different inputs do not seem to significantly affect the steady-state optimization selection gains for most benchmarks.

Figure 3.4 compares the performance of each program optimized with the best program-wide optimization phase set found by our genetic algorithm with the program performance achieved by the default HotSpot server compiler for both our benchmark suites. The error bars show the 95% confidence interval for the difference between the means over 10 runs of the best customized optimization selection and the default compiler sequence. We note that the default optimization sequence in the HotSpot compiler has been heavily tuned over several years to meet market expectations for Java performance, and thus presents a very aggressive baseline. In spite of this

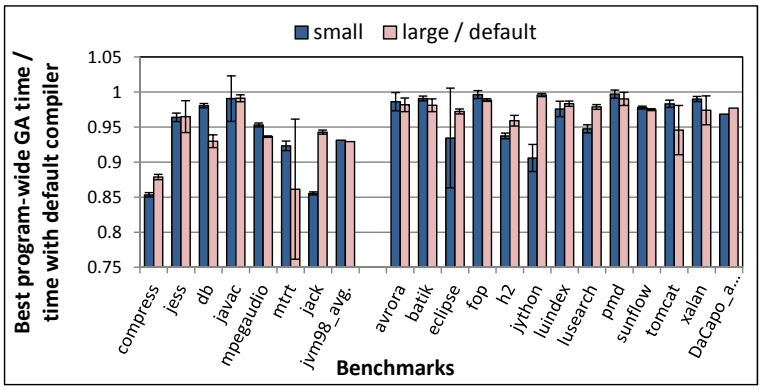


Figure 3.4: Performance of best program-wide optimization phase sequence after 100 generations of genetic algorithm.

aggressively tuned baseline, we find that customizing optimization selections can significantly improve performance (up to 15%) for several of our benchmark programs.

On average, the SPECjvm98 benchmarks improve by about 7% with both their *small* and *large* inputs. However, for programs in the DaCapo benchmark suite, program-wide optimization set specialization achieves smaller average benefits of 3.2% and 2.3% for their *small* and *default* input sizes respectively. The DaCapo benchmarks typically contain many more *hot* and *total* program methods as compared to the SPECjvm98 benchmarks. Additionally, unlike several SPECjvm98 programs that have a single or few dominant hot methods, most DaCapo benchmarks have a relatively flat execution profile with many methods that are similarly hot, with only slightly varying degrees (Blackburn et al., 2006). Therefore, program-wide optimization sets for DaCapo benchmarks are customized over much longer code regions (single optimization set over many more hot methods), which, we believe, results in lower average performance gains from program-wide optimization selection. Over all benchmarks, the average benefit of ideal program-wide phase selection is 4.3%.

3.5.3 Method-Specific Genetic Algorithm

The default HotSpot compiler optimizes individual methods at a time, and applies the same set of optimizations to each compiled method. Prior research has found that optimization phase se-

#	Benchmark	Method	% Time
1	db-large	Database.shell_sort	86.67
2	compress-small	Compressor.compress	54.99
3	compress-large	Compressor.compress	53.42
4	avrora-default	LegacyInterpreter.fastLoop	50.85
5	db-small	Database.shell_sort	50.72
6	jess-small	Node2.findInMemory	48.57
7	jack-small	TokenEngine.getNextTokenFromStream	48.05
8	avrora-small	LegacyInterpreter.fastLoop	44.49
9	jack-large	TokenEngine.getNextTokenFromStream	44.23
10	sunflow-default	KDTree.intersect	40.52
11	luindex-default	DocInverterPerField.processFields	40.43
12	sunflow-default	TriangleMesh$WaldTriangle.intersect	39.20
13	sunflow-small	KDTree.intersect	37.92
14	sunflow-small	TriangleMesh$WaldTriangle.intersect	36.78
15	jess-large	Node2.runTestsVaryRight	34.31
16	jython-small	PyFrame.getlocal	32.73
17	luindex-small	DocInverterPerField.processFields	30.51
18	lusearch-small	SegmentTermEnum.scanTo	29.88
19	lusearch-default	SegmentTermEnum.scanTo	28.76
20	jess-large	Node2.runTests	27.41
21	compress-large	Decompressor.decompress	24.86
22	compress-small	Compressor.output	23.39
23	mpegaudio-small	q.l	23.12
24	batik-default	MorphologyOp.isBetter	22.26
25	mpegaudio-large	q.l	21.87
26	jython-small	PyFrame.setline	21.79
27	xalan-small	ToStream.characters	21.70
28	db-small	ValidityCheckOutputStream.strip1	21.52
29	compress-large	Compressor.output	21.40
30	compress-small	Decompressor.decompress	21.23
31	xalan-default	ToStream.characters	20.00
32	pmd-default	DacapoClassLoader.loadClass	19.26
33	batik-small	PNGImageEncoder.clamp	17.74
34	sunflow-small	BoundingIntervalHierarchy.intersect	15.22
35	h2-small	Query.query	13.84
36	sunflow-default	BoundingIntervalHierarchy.intersect	13.79
37	javac-large	ScannerInputStream.read	13.46
38	javac-small	ScannerInputStream.read	13.17
39	luindex-small	TermsHashPerField.add	13.01
40	mpegaudio-small	tb.u0114	12.88
41	jython-default	PyFrame.setline	12.68
42	mpegaudio-large	tb.u0114	12.61
43	jess-large	Funcall.Execute	12.25
44	luindex-small	StandardTokenizerImpl.getNextToken	12.23
45	lusearch-small	IndexInput.readVLong	11.82
46	lusearch-default	StandardAnalyzer.tokenStream	11.49
47	lusearch-default	IndexInput.readVLong	11.46
48	lusearch-small	StandardAnalyzer.tokenStream	11.44
49	h2-default	Command.executeQueryLocal	11.37
50	luindex-default	TermsHashPerField.add	10.65
51	jython-default	PyFrame.getlocal	10.62
52	eclipse-default	Parser.parse	10.52
53	luindex-default	StandardTokenizerImpl.getNextToken	10.49

Table 3.2: Focus methods and their respective % of runtime

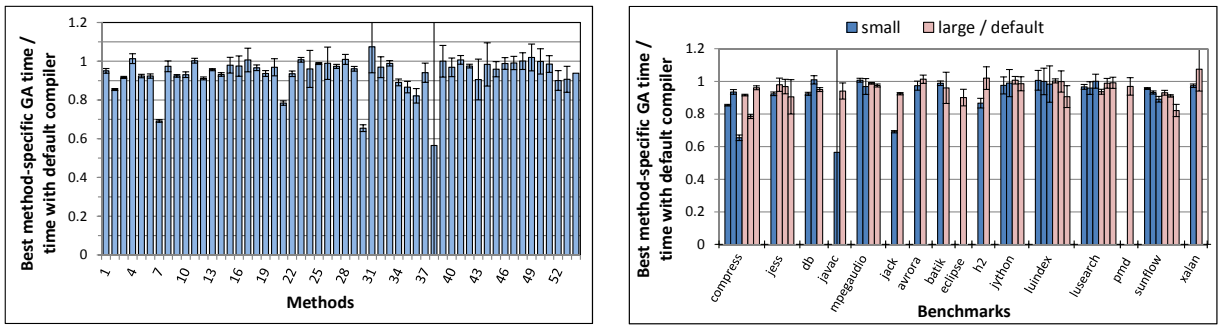
quences tuned to each method yield better program performance than a single program-wide phase sequence (Kulkarni et al., 2010; Agakov et al., 2006). In this section, we explore the performance potential of optimization selection at the method-level during dynamic JIT compilation.

3.5.3.1 Experimental Setup

There are two possible approaches for implementing GA searches to determine the performance potential of method-specific optimization phase settings: (a) running multiple simultaneous (and independent) GAs to gather optimization sequences for all program methods *concurrently* in the same run, and (b) executing the GA for each method *in isolation* (one method per program run).

The first approach requires instrumenting every program method to record the time spent in each method in a single program run. These individual method times can then be used to concurrently drive independent method-specific GAs for all methods in a program. The VM also needs the ability to use distinct optimization selections to be employed for different program methods. We implemented this experimental scheme for our HotSpot VM by updating the compiler to instrument each method with instructions that employ the x86 TSC (Time-Stamp Counter) to record individual method run-times. However, achieving accurate results with this scheme faces several challenges. The HotSpot JVM contains interprocedural optimizations, such as *method inlining*, due to which varying the optimization sequence of one method affects the performance behavior of other program methods. Additionally, we also found that the order in which methods are compiled can vary from one run of the program to the next, which affects optimization decisions and method run-times. Finally, the added instrumentation to record method times also adds some noise and impacts optimization application and method performance.

Therefore, we decided to employ the more straight-forward and accurate, but also time-consuming, approach of applying the GA to only one program method at a time. In each program run, the VM uses the optimization set provided by the GA to optimize one *focus method* and the default baseline set of optimizations to compile the other hot program methods. Thus, any reduction in the final program run-time over the baseline program performance can be attributed to the improvement in the single focus method. In an earlier *offline* run, we use our TSC based instrumentations with the baseline compiler configuration to estimate the fraction of total time spent by the program in each focus method. Any improvement in the overall program run-time during the GA is scaled with the fraction of time spent in the focus method to determine the run-time improvement in that individual method. We conduct this experiment over the 53 *focus methods* over all benchmarks that each comprise at least 10% of the time spent in their respective default program run. These methods, along with the % of total runtime each comprises in their respective benchmarks, are listed in Table 3.2.



(a) (b)

Figure 3.5: Performance of method-specific optimization selection after 100 GA generations. Methods in (a) are ordered by the % of run-time spent in their respective benchmarks. In (b), methods from the same benchmark are shown together. All results are are scaled by the fraction of total program time spent in the focus method and show the run-time improvement of that individual method.

3.5.3.2 Method-Specific GA Results

Figure 3.5(a) shows the *scaled* benefit in the run-time of each focus method when compiled with the best optimization set returned by the GA as compared to the method time if compiled with the baseline HotSpot server compiler. Methods along the x-axis in this graph are ordered by the fraction that they contribute to their respective overall program run-times (the same order methods are listed in Table 3.2). The final bar shows the average improvement over the 53 focus methods. Thus, we can see that customizing the optimization set for individual program methods can achieve significant performance benefits in some cases. While the best performance improvement is about 44%, method-specific optimization selection achieves close to a 6.2% reduction in run-time, on average. Figure 3.5(b) provides a different view of these same results with methods on the x-axis grouped together according to their respective benchmarks.

The plot in Figure 3.6 verifies that the (*non-scaled*) improvements in individual method run-times add-up over the entire program in most cases. That is, if individually customizing two methods in a program improves the overall program run-time by x and y respectively, then does the program achieve an $(x+y)$ percent improvement if both customized methods are used in the same program run? As mentioned earlier, our focus methods are selected such that each constitutes at least 10% of the baseline program run-time. Thus, different benchmarks contribute different

number (zero, one, or more than one) of focus methods to our set. The first bar in Figure 3.6 simply sums-up the individual method run-time benefits (from distinct program runs) for *benchmarks that provide two or more focus methods*. The second bar plots the run-time of code generated using the best customized optimization sets for all focus methods *in the same run*. We print the number of focus methods provided by each benchmark above each set of bars. Thus, we find that the individual method benefits add-up well in many cases, yielding performance benefit that is close to the sum of the individual benefit of all its customized component methods.

Please note that the experiments for Figure 3.6 only employ customized optimization selections for the focus methods. The remaining hot benchmark methods are compiled using the baseline sequence, which results in lower average improvements as compared to the average in Figure 3.5(a). Thus, customizing optimization sets over smaller program regions (methods (6.2%) vs. programs (4.3%)) realize better overall performance gains for JIT compilers.

It is important to note that we observe good correlation between the ideal method-specific improvements in Figure 3.5(a) and the per-method accumulated positive and negative impact of optimizations plotted in Figure 3.2. Thus, many methods with large accumulated negative effects (such as methods numbers #2, #3, #7, #21, #30, and #36) also show the greatest benefit from customized phase sequences found by our iterative (GA) search algorithm. Similarly, methods with small negative impacts in Figure 3.2 (including many methods numbered between #40 – #53) do not show significant benefits with ideal phase selection customization. While this correlation is encouraging, it may also imply that optimization interactions may not be very prominent in production-grade JVMs, such as HotSpot.

3.6 Effectiveness of Feature Vector Based Heuristic Techniques

Experiments in Sections 3.5.2 and 3.5.3 determine the potential gains due to effective phase selection in the HotSpot compiler. However, such iterative searches are extremely time-consuming, and are therefore not practical for dynamic compilers. Previous works have proposed using feature-

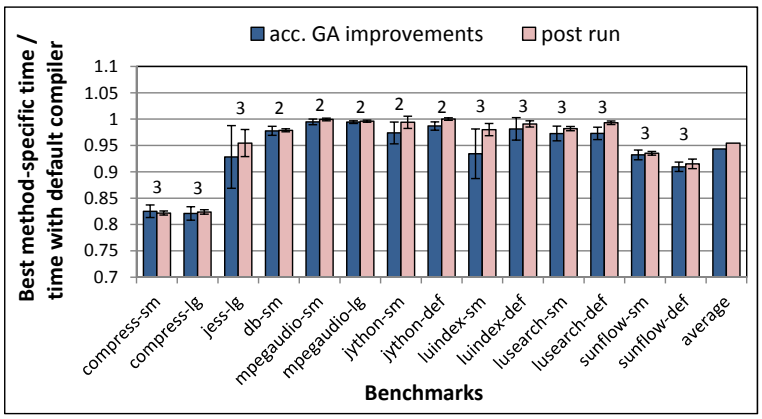


Figure 3.6: Accumulated improvement of method-specific optimization selection in benchmarks with multiple focus method.

vector based heuristic techniques to quickly derive customized optimization selections during online compilation (Cavazos & O'Boyle, 2006; Sanchez et al., 2011). Such techniques use an expensive *offline* approach to construct their predictive models that are then employed by a fast *online* scheme to customize phase selections to individual methods. In this section we report results of the first evaluation (compared to *ideal*) of the effectiveness of such feature-vector based heuristic techniques for finding good optimization solutions.

3.6.1 Overview of Approach

Feature-vector based heuristic algorithms operate in two stages, *training* and *deployment*. The training stage conducts a set of offline experiments that measure the program performance achieved by different phase selections for a certain set of programs. This stage then selects the best performing sets of phases for each method. The approach then uses a set of program *features* to characterize every compilation unit (method). The features should be selected such that they are representative of the program properties that are important and relevant to the optimization phases, and are easy and fast to extract at run-time. Finally, the training stage employs statistical techniques, such as logistic regression and support vector machines, to correlate good optimization phase settings with the method feature list.

The deployment stage installs the learned statistical model into the compiler. Now, for each

Scalar Features	Distribution Features	
Counters	**Types**	**ALU Operations**
Bytecodes	byte char	add sub
Arguments	int double	mul div
Temporaries	short long	rem neg
Nodes	float object	shift or
	address	and xor
		inc compare
Attributes	**Casting**	**Memory Operations**
Constructor	to byte	load load const
Final	to char	store new
Protected	to short	new array / multiarray
Public	to int	
Static	to long	**Control Flow**
Synchronized	to float	branch call
Exceptions	to double	jsr switch
Loops	to address	
	to object	**Miscellaneous**
	cast check	instance of throw
		array ops field ops
		synchronization

Table 3.3: List of method features used in our experiments

new compilation, the algorithm first determines the method's feature set. This feature set is given to the model that returns a customized setting for each optimization that is expected to be effective for the method. Thus, with this technique, each method may be compiled with a different phase sequence.

3.6.2 Our Experimental Configuration

We use techniques that have been successfully employed in prior works to develop our experimental framework (Cavazos & O'Boyle, 2006; Sanchez et al., 2011). Table 3.3 lists the features we use to characterize each method, which are a combination of the features employed in earlier works and those relevant to the HotSpot compiler. These features are organized into two sets: *scalar features* consist of counters and binary attributes for a given method without any special relationship; *distribution features* characterize the actual code of the method by aggregating similar operand types and operations that appear in the method. The *counters* count the number of bytecodes, arguments, and temporaries present in the method, as well as the number of nodes in the intermediate representation immediately after parsing. *Attributes* include properties denoted by keywords (*final*, *protected*, *static*, etc.), as well as implicit properties such as whether the method contains

loops or uses exception handlers. We record distribution features by incrementing a counter for each feature during bytecode parsing. The *types* features include Java native types, addresses (i.e. arrays) and user-defined objects. The remaining features correspond to one or more Java bytecode instructions. We use these features during the technique of logistic regression (Bishop, 1995) to learn our model for these experiments. Logistic regression has the property that it can even output phase sequences not seen during the model-training. We have tuned our logistic regression model to make it as similar as possible to the one used previously by Cavazos and O'Boyle (Cavazos & O'Boyle, 2006).

3.6.3 Feature-Vector Based Heuristic Algorithm Results

We perform two sets of experiments to evaluate the effectiveness of a feature-vector based logistic regression algorithm to learn and find good phase sequences for unseen methods during dynamic JIT compilation. As done in our other experiments, all numbers report the steady-state benchmark times. *All our experiments in this section employ cross-validation.* In other words, the evaluated benchmark or method (with both the small and large/default inputs) is never included in the training set for that benchmark or method.

Figure 3.7 plots the performance achieved by the optimization set delivered by the logistic regression algorithm when applied to each benchmark method as compared to the performance of the best benchmark-wide optimization sequence from Section 3.5.2. The training data for each program uses the top ten methods (based on their baseline run-times) from all the *other* (SPEC and DaCapo) benchmarks. While distinct benchmarks may contain different number of methods, we always consider ten methods from each program to weigh the benchmarks equally in the training set. For every benchmark, each top ten method contributes a distinct feature vector but uses the single benchmark-wide best optimization sequence from Section 3.5.2. The logistic regression algorithm may find a different optimization selection for each method during its online application. In spite of this flexibility, the feature vector based technique is never able to reach or improve the ideal single benchmark-wide optimization solution provided by the GA. Thus, figure 3.7 shows

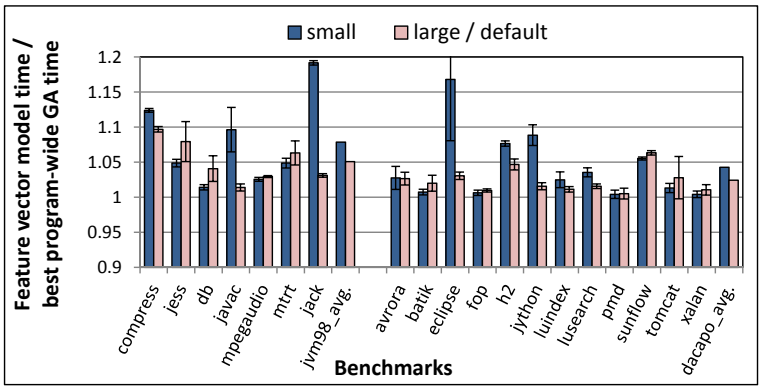


Figure 3.7: Effectiveness of benchmark-wide logistic regression. Training data for each benchmark consists of all the remaining programs from both benchmark suites.

that, on average, the feature-vector based solution produces code that is 7.8% and 5.0% worse for SPECjvm98 (small and large data sets respectively) and 4.3% and 2.4% worse for DaCapo (small and default) as compared to the ideal GA phase selection. However, this technique is some times able to find optimization sequences that achieve performances that are close to or better than those realized by the default HotSpot server compiler. On average, the feature-vector based heuristic achieves performance that is 2.5% better for SPECjvm98-small benchmarks, and equal in all other cases (SPECjvm98-large and DaCapo small and default) as compared to the *baseline server compiler*.

Figure 3.8 compares the performance of the logistic regression technique for individual program methods to their best GA-based performance. Since we employ cross-validation, the training data for each method uses information from all the other focus methods. Similar to the experimental configuration used in Section 3.5.3, each program run uses the logistic regression technique only for one *focus* method, while the remaining program methods are compiled with the baseline optimization sequence. The difference in program performance between the feature-vector based heuristic and the focus-method GA is scaled with the fraction of overall program time spent in the relevant method. Thus, we can see from Figure 3.8 that the per-method performance results achieved by the feature-vector based heuristic are quite disappointing. We find that, on average, the heuristic solutions achieve performance that is over 22% worse than the GA-tuned solution, and 14.7% worse than the baseline HotSpot server compiler.

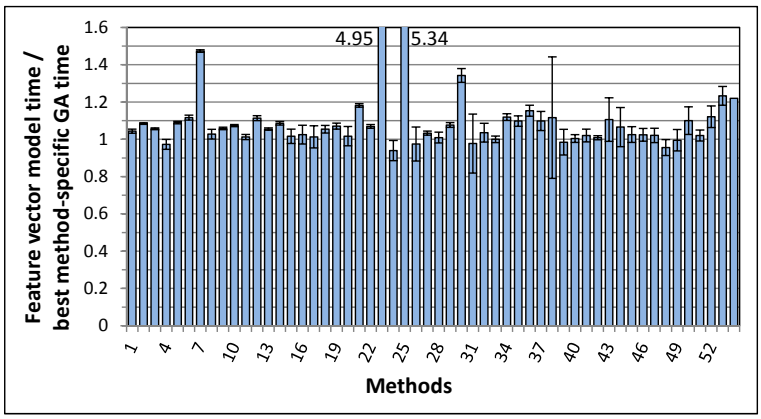
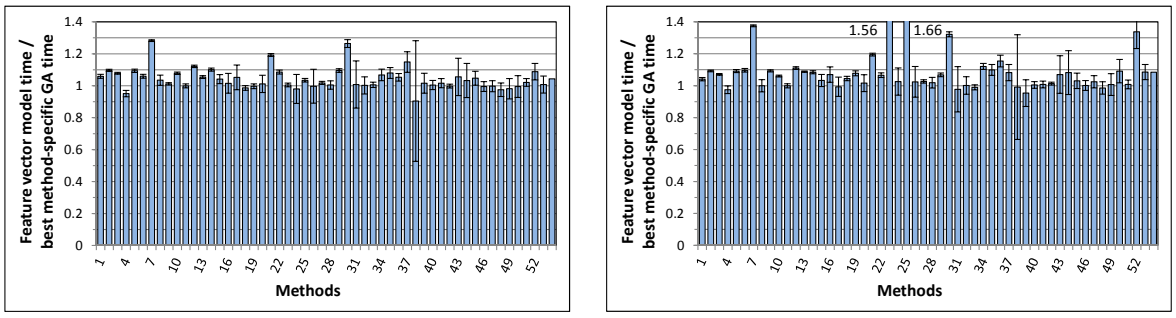


Figure 3.8: Effectiveness of method-specific logistic regression. Training data for each method consists of all the other focus methods used in Section 3.5.3.

3.6.4 Discussion

Thus, we find that existing state-of-the-art online feature-vector based algorithms are not able to find optimization sequences that improve code quality over the default baseline. We note that this observation is similar to the findings in other previous works (Cavazos & O'Boyle, 2006; Sanchez et al., 2011). However, these earlier works did not investigate whether this lack of performance gain is because optimization selection is not especially beneficial in online JIT compilers, or if existing feature-vector heuristics are not powerful enough to realize those gains. Our experiments conclusively reveal that, although modest on average, the benefits of optimization customization do exist for several methods in dynamic compilers. Thus, additional research in improving online phase selection heuristic algorithms is necessary to enable them to effectively specialize optimization settings for individual programs or methods. We conduct a few other experiments to analyze (and possibly, improve) the effectiveness of the logistic regression based feature-vector algorithm employed in this section.

In our first experiment we use the same per-method feature-vector based heuristic from the last section. However, instead of performing cross-validation, we allow the training data for each method to include that same method as well. Figure 3.9(a) plots the result of this experiment and compares the run-time of each method optimized with the phase selection delivered by the heuristic algorithm to the method's run-time when tuned using the ideal GA sequence. Thus,



(a) (b)

Figure 3.9: Experiments to analyze and improve the performance of feature-vector based heuristic algorithms for online phase selection. (a) Not using cross-validation and (b) Using observations for Section 3.4.

without cross-validation the heuristic algorithm achieves performance that is only 4.3% worse than ideal, and 2.5% *better* compared to the default HotSpot baseline. This result indicates that the logistic regression heuristic is not intrinsically poor, but may need a larger training set of methods, more subsets of methods in the training set with similar features that also have similar ideal phase sequences, and/or a better selection of method *features* to be more effective.

We have analyzed and observed several important properties of optimization phases in Section 3.4. In our next experiment, we employ the observation that most optimizations do not negatively impact a large number of methods to improve the performance of the feature-vector based heuristic (using cross-validation). With our new experiment we update the set of configurable optimizations (that we can set to ON/OFF) for each method to only those that show a negative effect on over 10% of the methods in the training set. The rest of the optimizations maintain their baseline ON/OFF configuration. Figure 3.9(b) shows the results of this experimental setup. Thus, we can see that the updated heuristic algorithm now achieves average performance that is 8.4% worse than ideal, and only 1.4% worse that the baseline.

There may be several other possible avenues to employ knowledge regarding the behavior and relationships of optimization phases to further improve the performance of online heuristic algorithms. However, both our experiments in this section show the potential and possible directions for improving the effectiveness of existing feature-vector based online algorithms for phase selection.

3.7 Future Work

There are multiple possible directions for future work. First, we will explore the effect of size and range of training data on the feature-vector based solution to the optimization selection problem for dynamic compilers. Second, we will attempt to improve existing heuristic techniques and develop new online approaches to better exploit the potential of optimization selection. In particular, we intend to exploit the observations from Section 3.4 and focus more on optimizations (and methods) with the most accumulated negative effects to build new and more effective online models. It will also be interesting to explore if more expensive phase selection techniques become attractive for the most important methods in later stages of *tiered* JIT compilers on multi-core machines. Third, we observed that the manner in which some method is optimized can affect the code generated for other program methods. This is an interesting issue whose implications for program optimization are not entirely clear, and we will study this issue in the future. Finally, we plan to repeat this study with other VMs and processor architectures to validate our results and conclusions.

3.8 Conclusions

The objectives of this research were to: (a) analyze and understand the phase selection related behavior of optimization phases in a production-quality JVM, (b) determine the steady-state performance potential of optimization selection, and (c) evaluate the effectiveness of existing feature-vector based heuristic techniques in achieving this performance potential and suggest improvements. We perform our research with the industry-standard Oracle HotSpot JVM to make our results generally and broadly applicable.

We found that most optimization phases in a dynamic JIT compiler only have a small effect on performance, and most phases do not negatively impact program run-time. These experiments also hinted at modest improvements by phase selection in dynamic JIT environments. Correspondingly, the GA-based *ideal* benchmark-wide and per-method optimization phase selection improves performance significantly in a few instances, but the benefits are modest on average (6.2% and 4.3%

for per-method and whole-program phase selection customization respectively). This result is not very surprising. To reduce compilation overhead, JIT compilers often only implement the more conservative optimization phases, which results in fewer optimizations and reduced, and possibly more predictable, phase interactions.

We also found that existing feature-vector based techniques used in dynamic compilers are not yet powerful enough to attain the ideal performance. We conducted experiments that demonstrate the directions for improving phase selection heuristics in the future. As part of this research, we have developed the first open-source framework for optimization selection research in a production-quality dynamic compilation environment. In the future, we expect this framework to enable further research to understand and resolve optimization application issues in JIT compilers.

Chapter 4

A Framework for Application Guidance in Virtual Memory Systems

This chapter proposes a collaborative approach in which applications can provide guidance to the operating system regarding allocation and recycling of physical memory. The operating system incorporates this guidance to decide which physical page should be used to back a particular virtual page. The key intuition behind this approach is that application software, as a generator of memory accesses, is best equipped to inform the operating system about the relative access rates and overlapping patterns of usage of its own address space. It is also capable of steering its own algorithms in order to keep its dynamic memory footprint under check when there is a need to reduce power or to contain the spillover effects from bursts in demand. Application software, working cooperatively with the operating system, can therefore help the latter schedule memory more effectively and efficiently than when the operating system is forced to act alone without such guidance. It is particularly difficult to achieve power efficiency without application guidance since power expended in memory is a function not merely of the intensity with which memory is accessed in time but also how many physical ranks are affected by an application's memory usage.

Our framework introduces an abstraction called "colors" for the application to communicate its intent to the operating system. We modify the operating system to receive this communication in an

efficient way, and to organize physical memory pages into intermediate level grouping structures called "trays" which capture the physically independent access channels and self-refresh domains, so that it can apply this guidance without entangling the application in lower level details of power or bandwidth management. This chapter describes how we re-architect the memory management of a recent Linux kernel to realize a three way collaboration between hardware, supervisory software, and application tasks.

4.1 Introduction

Recent trends in computer systems include an increased focus on power and energy consumption and the need to support multi-tenant use cases in which physical resources need to be multiplexed efficiently without causing performance interference. When multiplexing CPU, network, and storage facilities among multiple tasks a system level scheduling policy can perform fine-grained reassignments of the resources on a continuing basis to take into account task deadlines, shifting throughput demands, load-balancing needs and supply constraints. Many recent works address how best to allocate CPU time, and storage and network throughputs to meet competing service quality objectives (Wang et al., 2012; Lu et al., 2011), and to reduce CPU power during periods of low demand (Anagnostopoulou et al., 2012).

By comparison, it is very challenging to obtain precise control over distribution of memory capacity, bandwidth, or power, when virtualizing and multiplexing system memory. That is because these effects intimately depend upon how an operating system binds virtual addresses to physical addresses. An operating system uses heuristics that reclaim either the oldest, or the least recently touched, or a least frequently used physical pages in order to fill demands. Over time, after repeated allocations and reclaims, there is little guarantee that a collection of intensely accessed physical pages would remain confined to a small number of memory modules (or DIMMs). Even if an application reduces its dynamic memory footprint, its memory accesses can remain spread out across sufficiently many memory ranks to keep the ranks from saving much power. The layout of each

application's hot pages affects not just which memory modules can transition to lower power states during intervals of low activity, but also how much one program's activity in memory interferes with the responsiveness that other programs experience. Thus a more discriminating approach than is available in current systems for multiplexing of physical memory is highly desirable.

This chapter proposes a collaborative approach in which applications can provide guidance to the operating system in allocation and recycling of physical memory. This guidance helps the operating system take into account several factors when choosing which physical page should be used to back a particular virtual page. Thus, for example, an application whose high-intensity accesses are concentrated among a small fraction of its total address space can achieve power-efficient performance by guiding the operating system to co-locate the active pages among a small fraction of DRAM banks. Recent studies show that memory consumes up to 40% of total system power in enterprise servers (Lefurgy et al., 2003) making memory power a dominant factor in overall power consumption. Conversely, an application that is very intensive in its memory accesses may prefer that pages in its virtual address span are distributed as widely as possible among independent memory channels; and it can guide the operating system accordingly.

In order to provide these hints without entangling applications into lower level memory management details, we introduce the concept of coloring. Application software or middleware uses a number of colors to signal to the operating system a collection of hints. The operating system takes these hints into account during physical page allocation. Colors are applied against application selected ranges of virtual address space; a color is simply a concise way for an application to indicate to the operating system that some common behavior or intention spans those pages, even if the pages are not virtually contiguous. Colors can be applied at any time and can be changed as well, and the OS makes the best effort possible to take them into account when performing memory allocation, recycling, or page migration decisions. In addition to specializing placement, colors can also be used to signify memory priority or cache grouping, so that an OS can optionally support color-based displacement in order to implement software-guided affinitization of memory to tasks. This is particularly useful in consolidated systems where memory provisioning is important

for ensuring performance quality-of-service (QoS) guarantees.

Once an application colors various portions of its range with a few distinct colors, it tells the operating system what attributes (or combinations of attributes) it wants to associate with those colors. Depending upon how sophisticated an operating system implementation is, or what degrees of freedom are available to it in a given hardware configuration, the operating system tunes its allocation, displacement, migration, and power management decisions to take advantage of the information. At the same time, system properties and statistics that are necessary for the application to control its own behavior flow back from the OS, creating a closed feed-back loop between the application and host. That applications can guide the OS at a fine grained level in allocation and placement of pages is also an essential element of adapting applications to systems in which all memory is not homogeneous: for example, NVDIMMs, slower but higher capacity DIMMs, and secondary memory controllers may be used in an enterprise system to provide varying capabilities and performance. Mobile platforms represent another potential application as system-on-chip (SoC) designs are beginning to incorporate independently powered memory banks. The approach proposed in this chapter takes a critical first step in meeting the need for a fine-grained, power-aware, flexible provisioning of memory.

The physical arrangement of memory modules, and that of the channels connecting them to processors, together with the power control domains are all opaque to applications in our approach. In line with what has come to be expected from modern computer systems, our approach virtualizes memory and presents the illusion to every task that it has at its disposal a large array of memory locations. By hiding the lower level physical details from applications we preserve the benefits of modularity – applications do not need to become hardware aware in order to deliver memory management guidance to the operating system. Comparatively minor changes to the operating system bring about the necessary three way collaboration between hardware, supervisory software, and application tasks. We re-architect the memory management of a recent Linux kernel in order to achieve this objective. The OS reads the memory hardware configuration and constructs a software representation (called "trays") of all the power manageable memory units.

We used Linux kernel version 2.6.32 as a vehicle upon which to implement trays. We modified the kernel's page management routines to perform allocation and recycling over trays. We created application programming interfaces (APIs) and a suite of tools by which applications can monitor memory resources in order to implement coloring and associating intents with colors.

Our framework is the first to provide a system-level implementation of flexible, application-guided memory management. It supports such usage scenarios as prioritization of memory capacity and memory bandwidth, and saving power by transitioning more memory ranks into self-refresh states. It is easy to admit new scenarios such as application guided read-ahead or page prefill on a subset of ranges, differential placement of pages between fast and slow memory, and aggressive recycling of pages that applications can flag as transient. The major contributions of this work are:

- We describe in detail the design and implementation of our framework, which we believe is the first to provide for three way collaboration between hardware, supervisory software, and application tasks.

- We show how our framework can be used to achieve various objectives, including power savings, capacity provisioning, performance optimization, etc.

The next section places the contributions of this work in the context of related work and is followed by a description of relevant background information. We then describe the detailed design of our framework and present several experiments to showcase and evaluate potential use cases of application-guided memory management. We finally discuss potential future work before concluding the chapter.

4.2 Related Work

Researchers and engineers have proposed various power management schemes for memory systems. Bi et. al. (Bi et al., 2010) suggest predicting memory reference patterns to allow ranks to transition into low power states. Delaluz et. al. (Delaluz et al., 2002) track memory bank usage

in the operating system and selectively turn banks on and off at context switch points to manage DRAM energy consumption. Along these same lines, *memory compaction* has recently been integrated into the Linux kernel (Corbet, 2010). This technique, which reduces external fragmentation, is also used for power management because it allows memory scattered across banks to be compacted into fewer banks, in a way that is transparent to the application. Fan et. al. (Fan et al., 2003) employ petrinets to model and evaluate memory controller policies for manipulating multiple power states. Lin et. al. (Lin et al., 2009) construct an adaptive thread grouping and scheduling framework which assigns groups of threads to exclusive DRAM channels and ranks in order to jointly manage memory performance, power, and thermal characteristics. While these techniques employ additional hardware and system-level analysis to improve memory power management, our framework achieves similar objectives by facilitating collaboration between the application and host system.

Other works have explored integrating information at the application-level with the OS and hardware to aid resource management. Some projects, such as Exokernel (Engler et al., 1995) and Dune (Belay et al., 2012), attempt to give applications direct access to physical resources. In contrast to these works, our framework does not expose any physical structures or privileged instructions directly to applications. More similar to this work are approaches that enable applications to share additional information about their memory usage with lower levels of memory management. Prior system calls, such as *madvise* and *vadvise*, and various NUMA interfaces (Kleen, 2004) have allowed applications to provide hints to the memory management system. Some API's (such as Android's ashmem (Gonzalez, 2010) and Oracle's Transcendent Memory (Magenheimer et al., 2009)) allow the application or guest OS to allocate memory that may be freed by the host at any time (for instance, to reduce memory pressure). Banga, et. al. (Banga et al., 1999) propose a model and API that allows applications to communicate their resource requirements to the OS through the use of resource containers. Brown and Mowry (Brown & Mowry, 2000) integrate a modified SUIF compiler, which inserts *release* and *prefetch* hints using an extra analysis pass, with a runtime layer and simple OS support to improve response time of interactive applications in the presence

of memory-intensive applications. While these works evince some of the benefits of increased collaboration between applications and the OS, the coloring API provided by our framework enables a much broader spectrum of hints to be overlapped and provides a concise and powerful way to say multiple things about a given range of pages.

Also related is the concept of cache coloring (Kessler & Hill, 1992), where the operating system groups pages of physical memory (as the same *color*) if they map to the same location in a physically indexed cache. Despite their similar names, coloring in our framework is different than coloring in these systems. Cache coloring aims to reduce cache conflicts by exploiting spatial or temporal locality when mapping virtual pages to physical pages of different colors, while colors in our framework primarily serve to facilitate communication between the application and system-level memory management.

Finally, prior work has also explored virtual memory techniques for energy efficiency. Lebeck et. al. (Lebeck et al., 2000) propose several policies for making page allocation power aware. Zhou et. al. (Zhou et al., 2004) track the page miss ratio curve, i.e. page miss rate vs. memory size curve, as a performance-directed metric to determine the dynamic memory demands of applications. Petrov et. al. (Petrov & Orailoglu, 2003) propose virtual page tag reduction for low-power translation look-aside buffers (TLBs). Huang et. al. (Huang et al., 2003) propose the concept of power-aware virtual memory, which uses the power management features in RAMBUS memory devices to put individual modules into low power modes dynamically under software control. All of these works highlight the importance and advantages of power-awareness in the virtual memory system – and explore the potential energy savings. In contrast to this work, however, these systems do not provide any sort of closed-loop feedback between application software and lower level memory management, and thus, are vulnerable to learning inefficiencies as well as inefficiencies resulting from the OS and application software working at cross purposes.

4.3 Background

In order to understand the design and intuition of our framework, we first describe how current memory technologies are designed and viewed from each layer of the vertical execution stack, from the hardware up to the application.

Modern server systems employ a Non-Uniform Memory Access (NUMA) architecture which divides memory into separate regions (*nodes*) for each processor or set of processors. Within each NUMA node, memory is spatially organized into *channels*. Each channel employs its own memory controller and contains one or more DIMMs, which, in turn, each contain two or more *ranks*. Ranks comprise the actual memory storage and typically range from 2GB to 8GB in capacity. The memory hardware performs aggressive power management to transition from high power to low power states when either all or some portion of the memory is not active. Ranks are the smallest *power manageable unit*, which implies that transitioning between power states is performed at the rank level. Thus, different memory allocation strategies must consider an important power-performance tradeoff: distributing memory evenly across the ranks improves bandwidth which leads to better performance, while minimizing the number of active ranks consume less power.

The BIOS is responsible for reading the memory hardware configuration and converting it into physical address ranges used in the operating system. The BIOS provides several *physical address interleaving* configurations, which control how the addresses are actually distributed among the underlying memory hardware units. Different physical address interleaving configurations can have significant power-performance implications. For example, systems tuned for performance might configure the BIOS to fully interleave physical addresses so that consecutive addresses are distributed across all the available ranks.

On boot, the operating system reads the physical address range for each NUMA node from the BIOS and creates data structures to represent and manage physical memory. *Nodes* correspond to the physical NUMA nodes in the hardware. Each node is divided into a number of blocks called *zones* which represent distinct physical address ranges. Different zone types are suitable for different types of usage (e.g. the lowest 16MB of physical memory, which certain devices require,

is placed in the *DMA* zone). Next, the operating system creates physical page frames (or simply, *pages*) from the address range covered by each zone. Each page typically addresses 4KB of space. The kernel's physical memory management (allocation and recycling) operates on these pages, which are stored and kept track of on various lists in each zone. For example, a set of lists of pages in each zone called the *free lists* describes all of the physical memory available for allocation.

Finally, the operating system provides each process with its own virtual address space for managing memory at the application level. Virtual memory relieves applications of the need to worry about the size and availability of physical memory resources. Despite these benefits, virtualization adds a layer of abstraction between the application and operating system which makes it impossible for the lower-level memory management routines to derive the purpose of a particular memory allocation. Furthermore, even at the operating system level, many of the details of the underlying memory hardware necessary for managing memory at the rank level have been abstracted away as well. Thus, excessive virtualization makes it extremely difficult to design solutions for applications which require more fine-grained controls over memory resource usage.

4.4 Application-Guided Memory Management

This section describes the design and implementation of our framework. Enabling applications to guide management of memory hardware resources requires two major components:

1. An interface for communicating to the operating system information about how applications intend to use memory resources (usage patterns), and

2. An operating system with the ability to keep track of which memory hardware units (DIMMs, ranks) host which physical pages, and to use this detail in tailoring memory allocation to usage patterns

We address the first component in the next subsection, which describes our *memory coloring* framework for providing hints to the operating system about how applications intend to use mem-

ory resources. Next, we describe the architectural modifications we made to the system-level memory management software to enable management of individual memory hardware units.

4.4.1 Expressing Application Intent through Colors

A color is an abstraction which allows the application to communicate to the OS hints about how it is going to use memory resources. Colors are sufficiently general as to allow the application to provide different types of performance or power related usage hints. In using colors, application software can be entirely agnostic about how virtual addresses map to physical addresses and how those physical addresses are distributed among memory modules. By coloring any N different virtual pages with the same color, an application communicates to the OS that those N virtual pages are alike in some significant respect, and by associating one or more attributes with that color, the application invites the OS to apply any discretion it may have in selecting the physical page frames for those N virtual pages. As one rather extreme but trivial example, suppose that the application writer uses one color for N virtual pages and then binds with that color a guidance to "use no more than 1 physical page frame" for that color. The OS, if it so chooses, can satisfy a demand fault against any of those N page addresses simply by recycling one physical page frame by reclaiming it from one mapping and using it for another. Or, more practically, the OS may simply interpret such a guidance to allocate normally but then track any page frames so allocated, and reclaim aggressively so that those particular page frames are unlikely to remain mapped for long. A scan-resistant buffer-pool manager at the application level may benefit from this kind of guidance to the operating system.

More generally, an application can use colors to divide its virtual pages into groups. Each color can be used to convey one or more characteristics that the pages with that color share. Contiguous virtual pages may or may not have the same color. In this way an application provides a usage map to the OS, and the OS consults this usage map in selecting an appropriate physical memory scheduling strategy for those virtual pages. An application that uses no colors and therefore provides no guidance is treated normally– that is, the OS applies some default strategy. And even

when an application provides extensive guidance through coloring, depending on the particular version of the operating system, the machine configuration (such as how much memory and how finely interleaved it is), and other prevailing run time conditions in the machine, the OS may veer little or a lot from a default strategy. The specializations that an OS may support need not be confined just to selection of physical pages to be allocated or removed from the application's resident set, and they may include such other options as whether or not to fault-ahead or to perform read-aheads or flushing writes; whether or not to undertake migration of active pages from one set of memory banks to another in order to squeeze the active footprint into fewest physical memory modules. In this way, an OS can achieve performance, power, I/O, or capacity efficiencies based on guidance that application tier furnishes through coloring.

Using colors to specify intents instead of specifying intents directly (through a system call such as `madvise`) is motivated by three considerations- (a) efficiency through conciseness – an application can create the desired mosaic of colors and share it with the OS, instead of having to specify it a chunk at a time, and (b) the ability to give hints that say something horizontal across a collection of pages with the same color, such as, "it is desirable to perform physical page re-circulation among this group of virtual addresses", or, "it is desirable to co-locate the physical pages that happen to bind to this set of virtual addresses", etc., and (c) modularity – the capabilities supported by a particular version of an OS or a particular choice of hardware and system configuration may not support the full generality of hints that another version or configuration can support. An application developer or deployer, or some software tool, can bind colors to the menu of hints at load time or run time. This flexibility also means that even at run time, colors can be altered on the basis of feedback from the OS or guidance from a performance or power tuning assistant. In our prototype implementation, colors are bound to hints by a combination of configuration files and library software. A custom system call actually applies colors to virtual address ranges.

Let us illustrate the use of colors and hints with a simple example. Suppose we have an application that has one or more address space extents in which memory references are expected to be relatively infrequent (or uniformly distributed, with low aggregate probability of reference). The

application uses a color, say *blue* to color these extents. At the same time, suppose the application has a particular small collection of pages in which it hosts some frequently accessed data structures, and the application colors this collection *red*. The coloring intent is to allow the operating system to manage these sets of pages more efficiently – perhaps it can do so by co-locating the *blue* pages on separately power-managed units from those where *red* pages are located, or, co-locating *red* pages separately on their own power-managed units, or both. A possible second intent is to let the operating system page the *blue* ranges more aggressively, while allowing pages in the *red* ranges an extended residency time. By locating *blue* and *red* pages among a compact group of memory ranks, an operating system can increase the likelihood that memory ranks holding the *blue* pages can transition more quickly into self-refresh, and that the activity in *red* pages does not spill over into those ranks. (Other possible tuning options may be desirable as well – for example, allowing a higher clock frequency for one set of memory ranks and reducing clock frequency on others, if that is supported by the particular OS-hardware mix). Since many usage scenarios can be identified to the operating system, we define "intents" and specify them using configuration files. A configuration file with intents labeled *MEM-INTENSITY* and *MEM-CAPACITY* can capture two intentions: (a) that red pages are hot and blue pages are cold, and (b) that about 5% of application's dynamic resident set size (RSS) should fall into red pages, while, even though there are many blue pages, their low probability of access is indicated by their 3% share of the RSS.

1. Alignment to one of a set of standard intents:

   ```
   INTENT MEM-INTENSITY
   ```

2. Further specification for containing total spread:

   ```
   INTENT MEM-CAPACITY
   ```

3. Mapping to a set of colors:

   ```
   MEM-INTENSITY RED 0
   ```
   //hot pages

`MEM-CAPACITY RED 5` //hint- 5% of RSS

`MEM-INTENSITY BLUE 1` //cold pages

`MEM-CAPACITY BLUE 3` //hint- 3% of RSS

Next let us give an overview of the approach we have taken in implementing a prototype memory management system for receiving and acting upon such guidance. In our implementation, we have organized memory into power-management domains that are closely related to the underlying hardware. We call these management units *trays*, and we map colors to trays and tray based memory allocation and reclaim policies, as described in the following section.

4.4.2 Memory Containerization with Trays

We introduce a new abstraction called "trays" to organize and facilitate memory management in our framework. A *tray* is a software structure which contains sets of pages that reside on the same power-manageable memory unit. Each zone contains a set of trays and all the lists used to manage pages on the zone are replaced with corresponding lists in each tray. Figure 4.1 shows how our custom kernel organizes its representation of physical memory with trays in relation to the actual memory hardware.[1]

We have used Linux kernel version 2.6.32 as the baseline upon which to implement trays. The kernel's page management routines, which operate on lists of pages at the zone level were modified quite easily no operate over the same lists, but at a subsidiary level of trays. That is, zones are subdivided into trays, and page allocation, scanning, recycling are all performed at the tray level. While most of these changes are straightforward, the breadth of routines that require modification make the size of our patch substantial. (On last accounting, our kernel patch included modifications to approximately 1,800 lines of code over 34 files). This approach has the advantage that trays are defined as objects with attributes on each zone, which we find easier to reason about and maintain than another approach we considered: implicitly defining a "tray dimension" on

[1]The page interleaving shown in Figure 1 is for pictorial simplicity and is not a restriction.

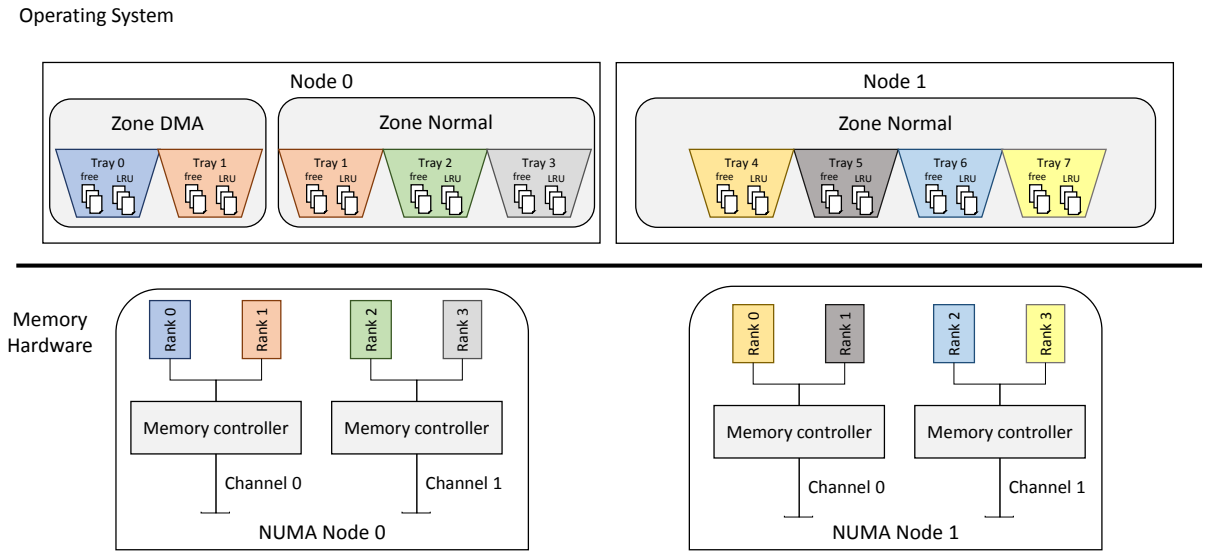


Figure 4.1: Physical memory representation in the Linux kernel with trays as it relates to the system's memory hardware.

each of the memory manager's lists of pages. Finally, this design requires less additional space overhead compared to other approaches. In contrast to the memory region approach proposed by A. Garg (http://lwn.net/Articles/445045/, 2011), which duplicates the entire zone structure for each memory hardware unit, the only structures we duplicate are pointers to list heads for each list of pages in each tray.

Assigning pages to the appropriate tray requires a mapping from the physical addresses served up by the BIOS to the individual memory hardware units. The ACPI 5.0 specification defines a memory power state table (MPST) which exposes this mapping in the operating system (http://www.acpi.info/spec 2011). Unfortunately, at the time of this writing, ACPI 5.0 has only been recently released, and we are not able to obtain a system conforming to this specification. Therefore, as a temporary measure, in our prototype implementation we construct the mapping manually from a knowledge of the size and configuration of memory hardware in our experimental system. Specifically, we know that each memory hardware unit stores the same amount of physical memory (2GB, in our case). We can also configure the BIOS to serve up physical addresses sequentially (as opposed to interleaving addresses across memory hardware units). Now, we can compute physical address

boundaries for each tray in our system *statically* using the size of our memory hardware units. Finally, at runtime, we simply map each page into the appropriate tray via the page's physical address. Note that, while, in our system, this mapping is simple enough that it does not require any additional per-page storage, a more complex mapping might require storage of tray information on each page after computing it once during initialization. Our immediate future work will be to rebase the implementation on an ACPI 5.0 compliant kernel that has the MPST information available to the Linux kernel at the point of handoff from BIOS.

4.5 Experimental Setup

This work presents several experiments to showcase and evaluate our framework's capabilities. In this section, we describe our experimental platform as well as the tools and methodology we use to conduct our experiments, including how we measure power and performance.

4.5.1 Platform

All of the experiments in this chapter were run on an Intel 2600CP server system with two Intel Xeon E5-2680 sockets (codename "Sandy Bridge"). Each socket has 8 2.7GHz cores with hyper-threading enabled and 16 GB of DDR3 memory (for a total of 32 threads and 32 GB of memory). Memory is organized into four channels per socket and each channel contains exactly one DIMM (with two 2 GB ranks each). We install 64-bit SUSE Linux Enterprise Server 11 SP 1 and select a recent Linux kernel (version 2.6.32.59) as our default operating system. The source code of this Linux kernel (available at `kernel.org`) provides the basis of our framework's kernel modifications.

4.5.2 The HotSpot Java Virtual Machine

Several of our experiments use Sun/Oracle's HotSpot Java Virtual Machine (build 1.6.0_24) (Paleczny et al., 2001). The latest development code for the HotSpot VM is available through Sun's OpenJDK

initiative. The HotSpot VM provides a large selection of command-line options, such as various JIT compilation schemes and different garbage collection algorithms, as well as many configurable parameters to tune the VM for a particular machine or application. For all of our experiments with HotSpot, we select the default configuration for server-class machines (http://www.oracle.com/technetwork/java/ja whitepaper 150215.pdf, 2006). We conduct our HotSpot VM experiments over benchmarks selected from two suites of applications: SPECjvm2008 (SPEC2008, 2008) and DaCapo-9.12-bach (Blackburn et al., 2006). Each suite employs a *harness* program to load and iterate the benchmark applications multiple times in the same run. The SPECjvm2008 harness continuously iterates several benchmark operations for each run, starting a new operation as soon as a previous operation completes. Each run includes a warmup period of two minutes (which is not counted towards the run score) and an iteration period of at least four minutes. The score for each run is the average time it takes to complete one operation during the iteration period. Similarly, the DaCapo harness executes each benchmark operation a specified number of iterations per run. For these applications, we execute the benchmark a total of 31 iterations per run and take the median runtime of the final 30 iterations as the score for each run. For all of our experiments, we report the average score of ten runs as the final score for the benchmark. We discuss the particular applications we use as well as further details about the HotSpot VM relevant to each experiment in the subsequent experimental sections.

4.5.3 Application Tools for Monitoring Resources

We designed several tools based on custom files in the `/proc` filesystem to provide applications with memory resource usage information. These tools provide an "on-demand" view of the system's memory configuration as well as an estimate of the total and available memory for each memory hardware unit (rank). Additionally, we have written tools to map specific regions of a process' virtual address space to the memory units backing these regions. These tools are essential for applications which require feedback from the OS to control their own memory usage.

4.5.4 DRAM Power Measurement

In addition to basic timing for performance measurements, many of our experiments read various activity counters (registers) to measure the impact of our framework with additional metrics, such as power consumption. We estimate DRAM power consumption with Intel's *power governor* library. The power governor framework employs sampling of energy status counters as the basis of its power model. The accuracy of the model, which is based on proprietary formulae, varies based on the DRAM components used. For our experiments, we run a separate instance of a power governor based-tool during each run to compute a DRAM power estimate (in Watts) every 100ms. We report the mean average of these estimates as the average DRAM power consumption over the entire run.

4.6 Emulating the NUMA API

Modern server systems represent and manage memory resources at the level of individual NUMA nodes. These systems typically provide a library and/or set of tools for applications to control how their memory usage is allocated across NUMA nodes. In contrast, our framework enables the operating system to manage resources at the more fine-grained level of individual hardware units. While our system provides more precise control, it is also powerful enough to emulate the functionality of tools designed specifically for managing memory at the level of NUMA node. In this section, we demonstrate that our framework effectively emulates the NUMA API by implementing a NUMA optimization in the HotSpot JVM with our framework.

4.6.1 Exploiting HotSpot's Memory Manager to improve NUMA Locality

The HotSpot JVM employs *garbage collection* (GC) to automatically manage the application's memory resources. The HotSpot memory manager allocates space for the application heap during initialization. As the application runs, objects created by the application are stored in this space. Periodically, when the space becomes full or reaches a preset capacity threshold, the VM runs a

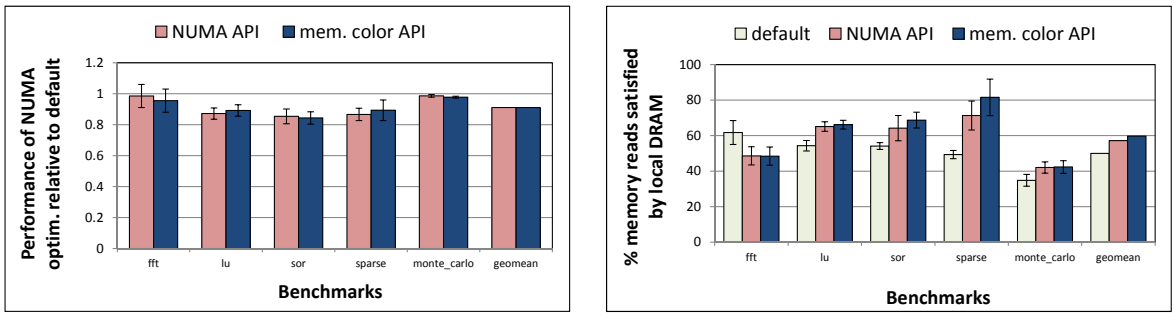
collection algorithm to free up space occupied by objects that are no longer reachable. The HotSpot garbage collector is *generational*, meaning that HotSpot divides the heap into two different areas according to the age of the data. Newly created objects are placed in an area called the *eden space*, which is part of the younger generation. Objects that survive some number of young generation collections are eventually promoted, or tenured, to the old generation.

On NUMA systems, this generational organization of memory can be exploited to improve DRAM access locality. The critical observation is thus: *newer objects are more likely to be accessed by the thread that created them*. Therefore, binding new objects to the same NUMA node as the thread that created them should reduce the proportion of memory accesses on remote NUMA nodes, and consequently, improve performance. In order to implement this optimization, HotSpot employs the NUMA API distributed with Linux. During initialization, HotSpot divides the eden space into separate virtual memory areas and informs the OS to back each area with physical memory on a particular NUMA node via the `numa_tonode_memory` library call. As the application runs, HotSpot keeps track of which areas are bound to which NUMA node and ensures that allocations from each thread are created in the appropriate area.

To implement the NUMA optimization in our framework, we create separate *node affinitization* colors for each NUMA node in the system. Next, we simply replace each call to `numa_tonode_memory` in HotSpot with a call to color each eden space area with the appropriate node affinitization color. The OS interprets this color as a hint to bind allocation for the colored pages to the set of trays corresponding to memory hardware units on a particular NUMA node. In this way, the node affinitization colors emulate the behavior of the `numa_tonode_memory` library call.

4.6.2 Experimental Results

We conduct a series of experiments to verify that our framework effectively implements the NUMA optimization in HotSpot. For these experiments, we select the SciMark 2.0 subset of the SPECjvm2008 benchmarks with the large input size. This benchmark set includes five computational kernels common in scientific and engineering calculations and is designed to address the performance of the



(a) (b)

Figure 4.2: Comparison of implementing the HotSpot NUMA optimization with the default NUMA API vs. our memory coloring framework (a) shows the performance of each implementation relative to the default HotSpot performance. (b) shows the % of NUMA-local memory reads with each configuration.

memory subsystem with out-of-cache problem sizes (SPEC2008, 2008). We found that enabling the HotSpot NUMA optimization significantly affects the performance of several of these benchmarks, making them good candidates for these experiments. We use the methodology described in Section 4.5.2 to measure each benchmark's performance. We also employ activity counters to measure the ratio of memory accesses satisfied by NUMA-local vs. NUMA-remote DRAM during each run.

Figure 4.2(a) shows the performance of each implementation of the NUMA optimization relative to the default configuration's performance of each benchmark. In Figure 4.2(a), lower bars imply better performance (e.g., the *lu* benchmark runs about 15% faster with the NUMA optimization enabled compared to the default HotSpot configuration). Also, for each result, we plot 95% confidence intervals using the methods described by Georges et. al. (Georges et al., 2007), and the rightmost bar displays the average (geometric mean) of all the benchmarks. It can be observed that the NUMA optimization improves performance for three of the five SciMark benchmarks, yielding about a 9% average improvement. More importantly, the performance results for each implementation of the NUMA optimization are very similar across all the benchmarks, with the differences between each implementation always within the margin of error. On average, the two implementations perform exactly the same. Similarly, Figure 4.2(b) plots the percentage of total memory accesses satisfied by NUMA-local memory for each configuration. As expected, the op-

timization increases the proportion of memory accesses satisfied by local DRAM for most of the benchmarks. Interestingly, the percentage of local memory accesses for one benchmark (*fft*) actually reduces with the optimization enabled. However, in terms of emulating the NUMA API, while these results show slightly more variation between the two implementations, the differences, again, are always within the margin of error. Thus, our framework provides an effective implementation of the NUMA optimization in HotSpot. Furthermore, this experiment shows that our design is flexible enough to supersede tools which exert control over memory at the level of NUMA nodes.

4.7 Memory Priority for Applications

Our framework provides the architectural infrastructure to enable applications to guide memory management policies for all or a portion of their own memory resources. In this section, we present a "first-of-its-kind" tool which utilizes this infrastructure to enable memory prioritization for applications. Our tool, called *memnice*, allows applications to prioritize their access to memory resources by requesting alternate management policies for low priority memory. Later, we showcase an example of using *memnice* to prioritize the memory usage of a Linux kernel compile.

4.7.1 *memnice*

Operating systems have long had the ability to prioritize access to the CPU for important threads and applications. In Unix systems, processes set their own priority relative to other processes via the *nice* system call, and the process scheduler uses each process' nice value to prioritize access to the CPU. Prioritizing access to memory resources is much more difficult due to the varying time and space requirements for memory resources and the layers of abstraction between the application and memory hardware. Our framework facilitates solutions for each of these issues enabling us to create the first-ever tool for prioritizing access to memory resources: *memnice*.

In our initial implementation, *memnice* accomplishes memory prioritization by limiting the

fraction of physical memory against which allocations can happen[2]. Consider a scenario in which several low-priority applications and one high-priority application compete over the same pool of memory. If left alone, the low-priority applications can expand their memory footprint contending with memory allocations of a higher-priority application, and forcing premature reclaims against a higher-priority application's pages.

In order to ensure the high-priority application runs smoothly, we can restrict the low-priority applications from using certain portions of memory using *memnice*. To implement this scheme in our framework, we enable applications to color portions of their address spaces to reflect the urgency with which pages must be stolen from elsewhere to meet demand fills. In the operating system, we restrict low-priority allocations to only search for free pages from an allowed set of trays. In this way, the low-priority applications are constrained to a smaller fraction of the system's memory, and the high-priority application can call upon larger fractions of system's memory during demand fills.

We showcase our implementation by running several Linux kernel compilations with different memory priority settings. Kernel compilation, unless restricted, can allocate and keep a large number of pages busy, as it proceeds through reading, creating, and writing, a lot of files, including temporary files, from each of a number of threads that run in parallel. For each experiment, we use the *memnice* utility, which is implemented as a simple command-line wrapper, to set the memory priority for the entire address space of the compilation process. We also configured our framework so that children of the root compilation process would inherit its memory priority and apply it to their own address spaces. We run all of our compilations on one node of the system described in Section 4.5.1. For each compilation, we employ four different memory priority configuration to restrict the compilation to a different set of memory resources in each run. These configurations restrict the compilation to use either 1 tray, 2 tray, 4 tray, or all 8 tray of memory on the node. Finally, during each run, we sampled (at a rate of once per second) the `proc` filesystem to estimate

[2]We chose a simple way to force prioritization. In a more refined implementation we would use colors to signal to the operating system that it should force a colored set to do color-local recycling subject to a minimum length of time for which pages are kept mapped before being recycled, to prevent thrashing

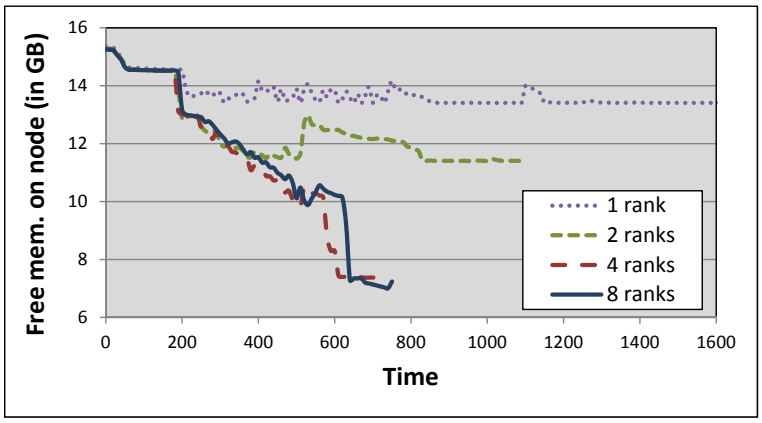


Figure 4.3: Free memory available during kernel compilations with different memory priorities

the amount of free memory available on the node.

4.7.2 Using *memnice* with Kernel Compilation

Each line in Figure 4.3 shows the free memory available on the node during kernel compilations with differing memory priorities. As we can see, each compilation proceeds at first using the same amount of memory resources. However, at around the 200th sample, the compilation restricted to use memory from only 1 tray is constrained from growing its memory footprint any larger than 2GB (recall that our system contains 2GB trays). Similarly, the compilation restricted to two trays stops growing after its footprint reaches about 4GB. The compilations restricted to 4 and 8 trays proceed in pretty much the same way for the entire run, presumably because kernel compilation does not require more than 8GB of memory. Finally, as expected, compilations restricted to use fewer memory resources take much longer to complete. Thus, *memnice* properly constrains the set of memory resources available for each application and is an effective tool for providing efficient, on-demand prioritization of memory resources.

4.8 Reducing DRAM Power Consumption

Memory power management in current systems occurs under the aegis of a hardware memory controller which transitions memory ranks into low power states (such as "self-refresh") during

periods of low activity. To amplify its effectiveness, it is desirable that pages that are very lightly accessed are not mixed up with pages that are accessed often within the same memory ranks. In this section, we show how our framework can bring about periods of low activity more consciously at a memory rank level, instead of relying on such an effect resulting from just happenstance.

For all the experiments in this section, we use a utility that we call *memory scrambler* to simulate memory state on a long-running system. As its name suggests, the"memory scrambler" is designed to keep trays from being wholly free or wholly allocated – it allocates chunks of memory until it exhausts memory, then frees the chunks in random order until a desired amount of memory is freed up (holding down some memory), effectively occupying portions of memory randomly across the physical address space. We perform the experiments described in this section using 16GB of memory from one socket of the server. Before each experiment, we run the scrambler, configured such that it holds down 4GB of memory after it finishes execution.

4.8.1 Potential of Containerized Memory Management to Reduce DRAM Power Consumption

The default Linux kernel views memory as a large contiguous array of physical pages, sometimes divided into NUMA nodes. That this array is actually a collection of independent power-managed ranks at the hardware level is abstracted away at the point that the kernel takes control of managing the page cache. Over time, as pages become allocated and reclaimed for one purpose after another, it is nearly impossible to keep pages that are infrequently accessed from getting mixed up with pages that are frequently accessed in the same memory banks. Thus, even if only a small fraction of pages are actively accessed, the likelihood remains high that they are scattered widely across all ranks in the system.

The use of *trays* in our customization of the default kernel makes it possible to reduce this dispersion. Trays complement coloring– the application guides the kernel so that virtual pages that are frequently accessed can be mapped to physical pages from one set of trays, while those that are not, can be mapped to physical pages from the remaining trays. If performance demand is

such that very high degree of interleaving is requested by an application, the operating system may need to spread out page allocation among more trays; but if that is not necessary then the operating system can keep page allocation biased against such a spread-out. In order to demonstrate the power-saving potential of our framework, we designed a "power-efficient" memory management configuration. We opt, when allocating a page, to choose a tray that has furnished another page for similar use. In this way we reduce the number of additional memory ranks that need to stay powered up.

To keep the first experiment simple, we have used a simplified workload. In this workload, we don't need application coloring because the workload simply allocates increasing amounts of memory in stages, so that in each stage it just allocates enough additional memory to fit in exactly one power-manageable unit. This unit is 2GB, in our case. In each stage, the workload continuously reads and writes the space it has allocated, together with all of the space it allocated in previous stages, for a period of 100 seconds. During each stage, we use power governor (Section 4.5.4) to measure the average DRAM power consumption.

We compare our custom kernel running the staged workload to the default kernel with two configurations: one with physical addresses interleaved across the memory hardware units (the default in systems tuned for performance) and another with physical addresses served up sequentially by the BIOS. Recall that our custom kernel requires that physical addresses are served up sequentially in order to correctly map trays onto power-manageable units.

Figure 4.4 shows the average DRAM power consumption during each stage for each system configuration. Thus, during stages when only a fraction of the total system memory is active, our custom kernel consumes much less power than the default kernel. Specifically, during the first stage, the custom kernel consumes about 55% less DRAM power than the default kernel with physical address interleaving enabled and 48% less than the default kernel with interleaving disabled. This is because, with no other processes actively allocating resources, the containerized kernel is able to satisfy the early stage allocations one memory unit at a time. The default kernel, however, has no way to represent power-manageable memory units and will activate memory on every

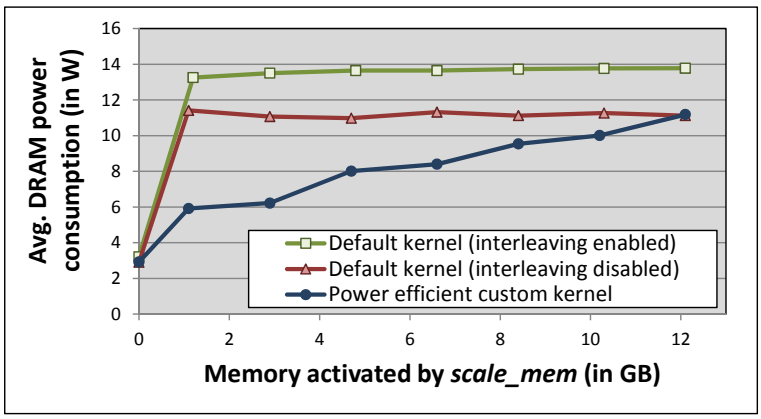


Figure 4.4: Relationship between memory utilization and power consumption on three different configurations

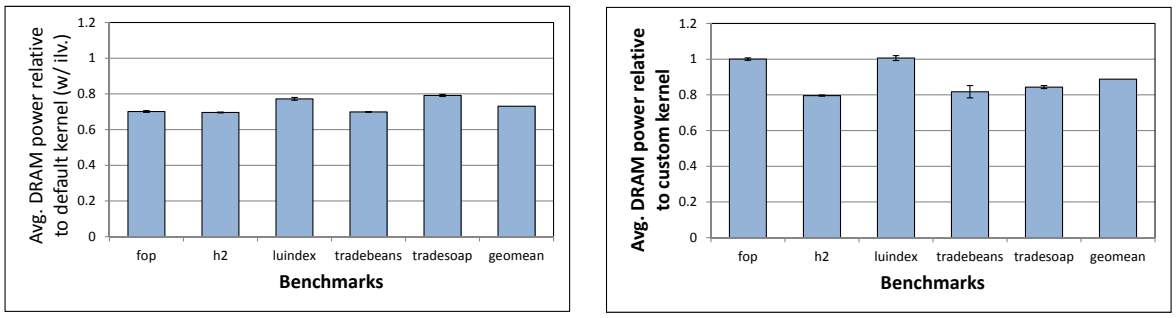
hardware unit during the first stage. As the workload activates more memory, the custom kernel activates more memory hardware units to accommodate the additional resources and eventually consumes as much DRAM power as the default kernel (with interleaving disabled).

In sum, these experiments show that our framework is able to perform memory management over power-manageable units and that this approach has the potential to significantly reduce DRAM power consumption.

4.8.2 Localized Allocation and Recycling to Reduce DRAM Power Consumption

In this section, we show that basic capabilities of our framework can be used to reduce DRAM power consumption over a set of benchmark applications. We create colors to enable applications to restrict their memory allocation and recycling to a population of pages that become allocated to each color (i.e., the OS performs color-local recycling). By restricting allocations of a given color to a subset of memory ranks, the operating system can translate color-confinement into confinement of the active memory footprints to the corresponding groups of ranks. We then measure the impact on power and performance from such confinements.

For these experiments, we selected five Java benchmarks from the DaCapo-9.12-bach benchmark suite for their varying memory usage characteristics. The *h2*, *tradebeans*, and *tradesoap*



(a) (b)

Figure 4.5: Local allocation and recycling reduces DRAM power consumption. (a) shows DRAM power relative to the default kernel (with interleaving enabled) and (b) shows the results relative to the custom kernel without local allocation and recycling.

applications execute thousands of transactions over an in-memory database and require relatively large memory footprints with non-uniform access patterns. In contrast, *fop*, which converts files to PDF format, and *luindex*, which indexes documents using the lucene library, require smaller footprints with more uniform access patterns.

We employ an experimental setup similar to the setup we use in the previous section. All experiments are run on one node of our server system and we again employ the memory scrambling tool to occupy random portions of the physical address space during each run. Each benchmark is run within the HotSpot JVM and we record performance and DRAM power measurements as described in Section 4.5. Each of the selected benchmarks requires no more than 4GB of total memory resources and we run the benchmarks one at a time. Thus, for each experimental run, we color the entire virtual address space of each benchmark to bind all memory allocation and recycling to a single 4GB DIMM.

Figure 4.5(a) shows the ratio of DRAM power consumed for each benchmark, between when it is run with local allocation and recycling and when it is run on the default kernel. As the figure shows, the local allocation and recycling consumes significantly less DRAM power than the default kernel configuration; indeed, DRAM power reduces by no less than 20% for each benchmark and the average savings are about 27%. It turns out, however, there are overlapping factors contributing to these power savings. The default configuration interleaves physical addresses across memory hardware units, which typically consumes more power than the alternative of non-interleaved ad-

dresses employed by our custom kernel configuration. Additionally, the custom kernel's tray based allocation may help by itself if the application's overall footprint is small enough. In order to isolate the effect of color-based allocation and recycling, we compare the average DRAM power consumed with and without color guidance with our custom kernel configuration (Figure 4.5(b)). We find that the database benchmarks are able to reduce power consumption based on color guidance, while the benchmarks with smaller memory footprints do not improve. This is because, without local allocation and recycling enabled, the database benchmarks scatter memory accesses across many power-manageable units, while the memory footprints of *fop* and *luindex* are small enough to require only one or two power-manageable units. Finally, it also appears that the potential reduction in memory bandwidth from localizing allocation and recycling to a single DIMM has very little impact on the performance of these benchmarks. We find that there is virtually no difference in performance between any of the configurations, including when compared to the default Linux kernel.

4.8.3 Exploiting Generational Garbage Collection

While it is useful to apply coarse-grained coloring over an application's entire virtual address space, let us use this section to describe the potential from using finer-grained control over portions of an application's address space, by evaluating in reality a proposal that has been previously explored through simulation.

The technique we evaluate in this section was first proposed and simulated by Velasco et. al. (Velasco et al., 2012) as a way to reduce DRAM power consumption in systems which employ generational garbage collection. Section 4.6.1 contained a description of the HotSpot garbage collector. The optimization (Velasco et al., 2012) exploits the observation that during young generation collection, only objects within the young generation are accessed. Thus, if objects in the tenured generation reside on an isolated set of power-manageable units (i.e. no other type of memory resides on them), then these units will power down during young generation collection. It is important to note that this optimization also relies on the assumption that the system "stops the

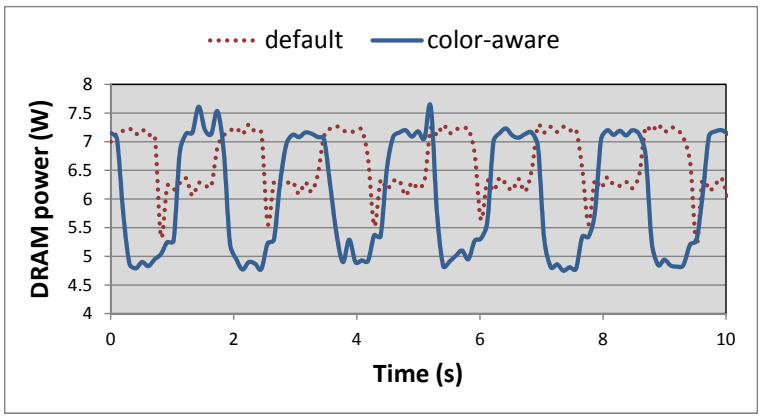


Figure 4.6: Raw power governor samples with and without "tenured generation optimization" applied

world" (i.e. does not allow application threads to run) during garbage collection to ensure that application threads do not access the tenured generation objects during young generation collection.

In order to implement this optimization in our framework, we arrange to house the tenured generation in a different set of power managed memory units than that allocated to the remainder, by modifying the HotSpot VM to color the tenured generation. When the operating system attempts to allocate resources for a virtual address colored as the tenured generation, it only searches for free memory from a subset of trays that it has earmarked for that color.

We ran the *derby* benchmark from the SPECjvm2008 benchmark suite for evaluation. *derby* is a database benchmark which spends a fair portion of its runtime doing garbage collection. We use the same machine and memory scrambler configuration that we use in the previous subsections and we measure performance and DRAM power consumption as described in Section 4.5.

Figure 4.6 plots the raw samples collected by the power governor tool during a portion of an experimental run with and without the optimization applied. Each dip corresponds to a separate invocation of the garbage collector. As we can see, the optimized configuration consumes slightly less power during garbage collection. Overall, the optimization reduces DRAM power by almost 9% and does not affect performance on our custom kernel. Thus, isolating the tenured generation memory in HotSpot on its own set of power-manageable units has a small but measurable impact on DRAM power consumption.

4.9 Future Work

Our current implementation of trays uses memory topology information that we provide to the operating system during its boot-up. This is a temporary measure; an immediate next task is to build upon MPST information that would be available in a Linux kernel that implements ACPI 5.0. Simultaneously we will implement color awareness in selected open source database, web server, and J2EE software packages, so that we can exercise complex, multi-tier workloads at the realistic scale of server systems with memory outlays reaching into hundreds of gigabytes. In these systems, memory power can reach nearly half of the total machine power draw, and therefore they provide an opportunity to explore dramatic power and energy savings from application-engaged containment of memory activities.

The experiments we reported in this chapter were on a small scale, in a system with just 16 or 32 gigabytes of memory as our first phase intent was to demonstrate the concept of application influenced virtualization of memory. In the next phase of this work, in addition to exploring the saving of power on a non-trivial scale of a terabyte data or web application services, we plan to explore memory management algorithms that permit applications to maximize performance by biasing placement of high value data so that pages in which performance critical data resides are distributed widely across memory channels. Another element of optimization we plan to explore is application guided read-ahead and/or fault-ahead for those virtual ranges of applications in which there is reason to expect sequential access patterns. We also plan to implement a greater variety of page recycling policies in order to take advantage of color hints from applications; for instance, we would like to create the possibility that trays implement such algorithm options as (a) minimum residency time– where a page is not reclaimed unless it has been mapped for a certain threshold duration, (b) capacity allocation – where, physical pages allocated to different color groups are recycled on an accelerated basis as needed in order to keep the time-averaged allocation in line with application guided capacities, (c) reserve capacities – in which a certain minimum number of pages are set aside in order to provide a reserve buffer capacity for certain critical usages, etc. We envision that as we bring our studies to large scale software such as a complex database, we will

inevitably find new usage cases in which applications can guide the operating system with greater nuance about how certain pages should be treated differently from others. Finally, one avenue of considerable new work that remains is the development of tools for instrumentation, analysis, and control, so that we can facilitate the generation and application of memory usage guidance between application software and operating system.

4.10 Conclusion

This chapter presents a framework for application-guided memory management. We create abstractions called colors and trays to enable three-way collaboration between the application tasks, operating system software, and hardware. Using a re-architected Linux kernel, and detailed measurements of performance, power, and memory usage, we demonstrate several use cases for our framework, including emulation of NUMA APIs, enabling memory priority for applications, and power-efficiency for important applications. Hence, by providing an empirical evaluation that demonstrates the value of application-guided memory management, we have shown that our framework is a critical first step in meeting the need for a fine-grained, power-aware, flexible provisioning of memory.

Chapter 5

Automatic Cross-Layer Memory Management to Reduce DRAM Power Consumption

Energy efficiency has become a major concern in a wide range of computing domains. It is difficult to achieve power efficiency in memory because the power expended in memory is not only a function of the intensity with which memory is accessed in time, but also how many physical memory devices (i.e. DRAM ranks) that are affected by an application's memory usage. To improve energy efficiency in the memory subsystem, we build a novel, collaborative, cross-layer, framework that employs object profiling and management at the application layer to enable power-efficient distribution of objects in the DRAM hardware. Our framework, based on Oracle's HotSpot Java Virtual Machine and the Linux operating system, automatically classifies and organizes program objects into separate heap regions based on their expected usage. The VM communicates the region boundaries and expected usage behavior to our modified Linux kernel, which incorporates this information during memory management to organize physical memory in a power-efficient way.

In this chapter, we describe the design and implementation of our framework, which is the first

to integrate object profiling and analysis at the application layer with fine-grained management of memory hardware resources in the operating system. We use a custom memory-intensive workload to demonstrate the potential of our approach. Next, we develop a new profiling-based analysis and code generator in the HotSpot VM to *automatically* determine and control the placement of hot and cold objects in a partitioned VM heap. This information is communicated to the operating system, which uses it to map the logical application pages to the appropriate DRAM ranks. We evaluate our framework and find that it achieves significant DRAM energy savings across a variety of workloads in the SPECjvm2008 benchmark suite, without any source code modifications or recompilations.

5.1 Introduction

Recent trends in computing include an increased focus on energy efficiency. Computer systems are becoming more power hungry, and researchers are actively searching for better ways to manage system power. At the platform level (individual node or server), most effort has focused on reducing power consumption in the CPU. However, recent research suggests that power consumption in memory has become a dominant factor for many server systems (Lefurgy et al., 2003). Not surprisingly, it is very challenging to obtain precise control over the distribution and usage of memory power or bandwidth when virtualizing system memory. These effects depend upon the assignment of virtual addresses to the application's data objects, the OS binding of virtual to physical addresses, and the mapping of physical pages to hardware DRAM devices.

To overcome these challenges, the previous chapter proposes a framework to facilitate communication across multiple layers of the vertical execution stack during memory management. This framework enables applications to communicate guidance to the OS about how they intend to use portions of the virtual address space. The operating system can then incorporate this guidance when deciding which physical page should be used to back a particular virtual page.

While this framework provides a powerful tool for communicating information from the ap-

plications to the OS during memory management, it suffers from some severe limitations: (a) For many complex applications and workloads, it will be infeasible to manually determine the set of memory usage guidance to provide to the OS, and (b) All memory usage guidance must be manually inserted into source code, and modified applications must be recompiled.

This chapter aims to address the limitations of such OS-level frameworks by integrating them with an *automated* mechanism to determine and convey memory access and usage information from the application to the OS at run-time without any additional recompilations. Our work, implemented in the standard HotSpot Java Virtual Machine (JVM), divides the application's *heap* into separate regions for objects with different (expected) usage patterns. At application run-time, we use profiling information in our custom JVM to *automatically* partition and allocate the heap objects into separate regions and communicate with the OS to guide memory management.

To motivate our approach, we design a custom benchmark that uses static program types to distinguish objects with different usage activity. We conduct a series of experiments and present detailed analysis that shows that controlling the placement of program objects yields DRAM energy savings of up to 30% with this memory-intensive workload. We then evaluate our fully automated profiling-based approach on the entire set of SPECjvm2008 benchmarks, and report our observations. This work makes the following important contributions:

1. We develop, implement, and evaluate a framework to automatically partition application data objects into distinct classes based on their usage patterns,

2. We design a custom benchmark to effectively demonstrate the potential and tradeoffs of memory power saving with program speed and program bandwidth requirements,

3. We build an exhaustive profiling framework in the HotSpot JVM to determine program memory usage patterns at run-time, and

4. We provide detailed experimental results and performance analysis of our cross-layer framework for memory management.

5.2 Related Work

Several researchers have explored the effect of object layouts on program performance. Hirzel (Hirzel, 2007) conducted an extensive evaluation to demonstrate the importance of data layout for program performance. Zhang and Hirzel (Zhang & Hirzel, 2008) employ layout auditing to automatically select data layouts that perform well. Jula and Rauchwerger (Jula & Rauchwerger, 2009) propose two memory allocators that use automatically provided allocation hints to improve spatial locality. Hirzel et. al. (Hirzel et al., 2003) and Guyer and Mckinley (Guyer & McKinley, 2004) manipulate object layout to implement optimizations in the garbage collector. In contrast to these works, which only affect object layout in the virtual address space, our framework controls the physical location of objects in memory as well.

Other works have explored integrating information at the application-level with the OS and hardware to aid resource management. Projects, such as Exokernel (Engler et al., 1995) and Dune (Belay et al., 2012), attempt to give applications direct access to physical resources. In contrast, our framework does not use or expose any physical structures or privileged instructions directly to applications. Banga, et. al. (Banga et al., 1999) propose a model and API that allows applications to communicate their resource requirements to the OS through the use of resource containers. Brown and Mowry (Brown & Mowry, 2000) integrate a modified SUIF compiler, which inserts *release* and *prefetch* hints using an extra analysis pass, with a runtime layer and simple OS support to improve response time of interactive applications in the presence of memory-intensive applications.

Also related is the concept of cache coloring (Kessler & Hill, 1992), where the operating system groups pages of physical memory (as the same *color*) if they map to the same location in a physically indexed cache. Despite their similar names, coloring in our framework is different than coloring in these systems. Cache coloring aims to reduce cache conflicts by exploiting spatial or temporal locality when mapping virtual pages to physical pages of different colors, while colors in our framework primarily serve to signal usage intents from the JVM to the operating system.

Prior work has also explored virtual memory techniques for energy efficiency. Lebeck et.

al. (Lebeck et al., 2000) propose several policies for making page allocation power aware. Zhou et. al. (Zhou et al., 2004) track the page miss ratio curve, i.e. page miss rate vs. memory size curve, as a performance-directed metric to determine the dynamic memory demands of applications. Petrov et. al. (Petrov & Orailoglu, 2003) propose virtual page tag reduction for low-power translation look-aside buffers (TLBs). Huang et. al. (Huang et al., 2003) propose the concept of power-aware virtual memory, which uses the power management features in RAMBUS memory devices to put individual modules into low power modes dynamically under software control. All of these works highlight the importance and advantages of power-awareness in the virtual memory system – and explore the potential energy savings. In contrast to our work, these systems do not employ integration between upper- and lower-level memory management, and thus, are vulnerable to learning inefficiencies as well as those resulting from the OS and application software working at cross purposes.

5.3 Background

In order to effectively improve memory power-efficiency, it is important to understand how memory devices are designed and used in modern computing systems. In this section, we present an overview of how memory hardware is organized in a typical server machine, discuss factors contributing to DRAM power consumption, as well as describe how memory is represented and managed by the operating system and applications.

5.3.1 Overview of DRAM Structure

In modern server systems, memory is spatially organized into *channels*. Each channel employs its own memory controller and contains one or more DIMMs, which, in turn, each contain two or more *ranks*. Ranks comprise the actual memory storage and typically range from 2GB to 8GB in capacity.

5.3.2 DRAM Power Consumption

DRAM power consumption can be divided into two main components: *operation power*, which is the power required for active memory operations, and *background power*, which accounts for all other power in the memory system[1] Operation power is primarily driven by the number of memory accesses to each device, while background power depends solely on the operating frequency and the current power-down state of the memory devices. Modern memory devices perform aggressive power management to automatically transition from high power to low power states when either all or some portion of the memory is not active. Ranks are the smallest *power manageable unit*, which implies that transitioning between power states is performed at the rank level. As with any power-management technique, turning off a portion of the device in a power-down state incurs a wakeup penalty the next time that portion of the device is accessed. Current DDR3 technology supports multiple power-down states, each of which poses a different tradeoff between power savings and wakeup latency. Table 3 in (David et al., 2011) provides a summary of power requirements and wakeup latencies for the various power-down states for a typical DDR3 device. Thus, controlling memory power consumption requires understanding how memory accesses are distributed across memory devices. Configurations tuned to maximize performance attempt to distribute accesses evenly across the ranks and channels to improve bandwidth and reduce wakeup latencies, while low-power configurations attempt to minimize the number of active memory devices.

5.3.3 Operating System View of Physical Memory

During system initialization, the BIOS reads the memory hardware configuration and converts it into physical address ranges provided to the operating system. Many vendors provide options to control (prior to boot) how physical addresses are distributed among the underlying memory hardware units. For example, options to enable channel and rank interleaving ensure that consecutive physical addresses are distributed across the system's memory devices. At boot time, the operating

[1]For a full accounting of the various factors that contribute to DRAM power consumption, see Section 3 in (David et al., 2011).

system reads the physical address range for each NUMA node from the BIOS and creates data structures to represent and manage physical memory. *Nodes* correspond to the physical NUMA nodes in the hardware. Each node is divided into a number of blocks called *zones* which represent distinct physical address ranges. Next, the operating system creates physical page frames (or simply, *pages*) from the address range covered by each zone. Page size varies depending on the system architecture and configuration, but in a typical x86 machine, each page addresses 4KB of memory. The kernel's physical memory management (allocation and recycling) operates on these pages, which are stored and kept track of on various lists in each zone. For example, a set of lists of pages in each zone called the *free lists* describes all of the physical memory available for allocation. Most current systems do not have any view of how physical pages map to the underlying memory topology (i.e. the actual layout of DRAM ranks and channels in hardware) during memory management.

5.3.4 Automatic Heap Management in Managed Language Runtimes

The operating system provides each process with its own virtual address space for using memory at the application level. Native applications use system calls (e.g. *brk* and *mmap*) to request and manage virtual memory resources. Programs written in managed languages (such as Java) do not interact directly with the operating system, but rather, execute inside a process virtual machine (VM) (also called runtime system). In addition to enabling portability, the VM provides various services, including garbage collection (GC), for developer convenience and to facilitate safer and more efficient computation. On initialization, the VM allocates a large virtual address space for use as the application's heap. When the application allocates an object (e.g. using Java's *new* instruction), the VM updates a pointer in the application's heap area to reserve space for the new object. Periodically, the heap area will fill with objects created by the application, triggering a garbage collection. During GC, the VM frees up space associated with the dead (unreachable) objects, and possibly shrinks or expands the virtual memory space depending on the current need. In this way, the VM automatically manages several aspects of the application's memory resources,

including the size of the application's heap and where program objects are located in the virtual memory space.

5.4 Cross-Layer Memory Management

Controlling the placement of program objects/data in memory to reduce memory power consumption requires collaboration between multiple layers of the vertical execution stack, including the application, operating system, and DRAM hardware. Our modified HotSpot JVM divides its application heap into separate regions for objects with different expected usage patterns. We integrate our modified JVM with the OS-based framework described in the previous chapter that provides two major components: (a) An application programming interface (API) for communicating to the OS information about how applications intend to use memory resources (usage patterns), and (b) An operating system with the ability to keep track of which memory hardware units (DIMMs, ranks) host which physical pages, and to use this detail in tailoring memory allocation to usage patterns.

Incorporating application guidance during memory management can enable the OS to achieve more efficient memory bandwidth, power, and/or capacity distributions. While the existing OS framework facilitates communication between the applications and operating system, it has some serious limitations:

1. It requires developers to manually determine the set of colors or coloring hints that will be most useful, which may be tedious or difficult for many applications, and

2. It requires all colors and coloring hints to be manually inserted into source code, and all colored applications must be recompiled.

This work aims to address these limitations by integrating application guidance with a custom Java virtual machine that *automatically* partitions program objects into separately colored regions. Our custom JVM divides the application's heap into multiple distinct regions, each of which is

colored to indicate how the application intends to use objects within the region. For example, in our initial implementation, we divide the application's heap into two regions: one for objects that are accessed relatively frequently (i.e. hot objects), and another for objects that are relatively cold. Colors are applied to each space during initialization and whenever region boundaries change due to heap resizing. As the application runs, the OS interprets colors supplied by the VM and attempts to select physical memory scheduling strategies tailored to the expected usage behavior of each region. The VM is free to allocate and move objects among the colored spaces, but it makes every effort possible to ensure that the objects reside in the appropriate space. Depending on the chosen coloring, this might require additional profiling and/or analysis of object usage patterns or program instructions in order to be effective. Fortunately, the VM provides a convenient environment for monitoring application behavior. Several VM tasks (such as selective compilation, code optimization, and heap resizing) already use profile feedback to help guide optimization decisions (Arnold et al., 2005).

We employ Oracle's HotSpot JVM to implement our framework. Details about our specific HotSpot version and how it is configured are provided in Section 5.6.3. The default garbage collector in HotSpot is *generational*, meaning that the GC divides the heap into different areas according to the age of the data. Newly created objects are placed in an area called the *eden space*, which is part of the younger generation. The survivor space holds objects that have survived at least one young generation collection, while objects that have survived a number of young generation GC's are eventually promoted, or tenured, to the older generation.

Figure 5.1 shows how our custom JVM framework organizes the application heap into separate spaces for hot and cold objects. Each space within each generation is divided into two equal-sized regions: one colored orange for hot (frequently accessed) objects and the other colored blue for cold objects. As the application executes, the VM attempts to allocate and move program objects into the appropriate colored space. For our initial implementation, we conduct a profile run to gather information about object access patterns and program allocation points that we use to guide object coloring decisions in the current run. We describe our profiling framework in Section 5.7.1.

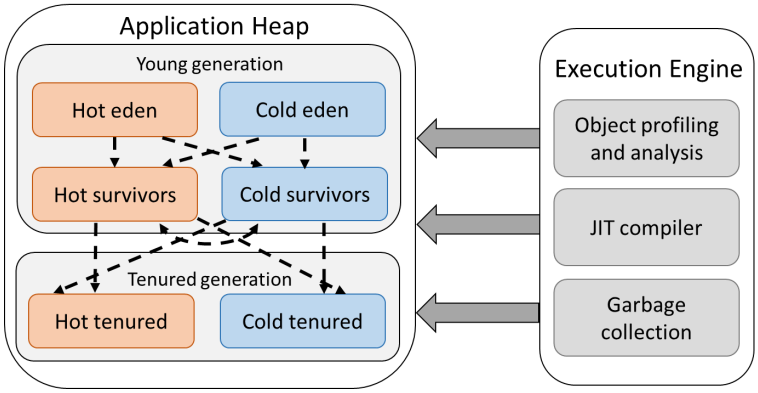


Figure 5.1: Colored spaces in our JVM framework. Dotted lines indicate possible paths for objects to move from one colored space to another.

The VM consults this profiling information at each object allocation point to determine which color to assign the new object. Periodically, perhaps at every garbage collection cycle, the VM may also re-assign colors for live objects depending on object profiling information and / or program phase behavior. Note, however, that while our framework is capable of dynamically reassigning colors for live objects, for all of the experiments in this work, object colors assigned at object allocation time are kept static throughout the entire program run.

5.5 Potential of Cross-Layer Framework to Reduce DRAM Energy Consumption

Memory power management in current systems occurs under the aegis of a hardware memory controller that transitions memory ranks into low power states (such as "self-refresh") during periods of low activity. To amplify its effectiveness, it is desirable that pages that are very lightly accessed are not located on the same memory ranks with pages that are accessed often. In this section, we demonstrate the potential of our framework to reduce DRAM energy consumption by controlling the placement of hot and cold objects across memory ranks. The experimental platform and tools we use to collect timing, DRAM bandwidth and power measurements are described in Section 5.6.

5.5.1 The MemBench Benchmark

To demonstrate the power-saving potential of our framework we construct a custom benchmark (called MemBench) that differentiates hot and cold objects *statically* by declaring objects with distinctive program types in its source code. The MemBench benchmark creates two main types of objects: *HotObject* and *ColdObject*. Each object contains an array of integers (comprising about 1MB of space per object) which represent the objects' memory resources. On initialization, MemBench allocates a large number of such objects and stores them in a single, large, in-memory array. A certain portion of the objects (about 15% for these experiments) are allocated as the HotObject type, while the remaining objects are allocated as the ColdObject type. The order of object allocations is randomized so that HotObjects and ColdObjects are stored intermittently throughout the object array. For each of our experiments, we configure MemBench to allocate 27GB of program objects (or approximately 90% of the free memory on our server node).

After MemBench is done allocating the hot and cold objects, the object array is divided into 16 roughly equal parts (one for each hardware thread on our server node) that are each passed to their own software thread. The threads (simultaneously) iterate over their own portion of the object array for some specified number of iterations, passing over the cold objects, and only accessing memory associated with the hot objects. When a hot object is reached, the thread performs a linear scan over the object's associated integer array, storing a new value into each cell as it walks the array. The benchmark completes when all threads have finished iterating over their portion of the object array. For these experiments, we select a large enough number of iterations so that each scanning thread runs for at least 100 seconds.

Additionally, in order to allow experiments with different memory bandwidth requirements, MemBench provides an optional delay parameter that can be configured to slow down the rate at which memory accesses occur. The delay is implemented by inserting additional computation between stores to the integer array associated with each hot object. Specifically, the delay mechanism iteratively computes a Fibonacci number between integer stores. Thus, computing a larger Fibonacci number imposes a longer delay between memory accesses, which reduces the average

Table 5.1: Allocation Time for the MemBench Benchmark.

Configuration	Time (s)
default	88.4
tray-based kernel	103.0
hot/cold organize	118.1

bandwidth requirements for the workload.

5.5.2 Experimental Evaluation

We constructed three configurations to evaluate our cross-layer framework with the MemBench benchmark. The *default* configuration employs the unmodified Linux kernel with the default HotSpot JVM. The *tray-based kernel* configuration runs MemBench with the unmodified HotSpot on the custom kernel framework Since the JVM in the tray-based kernel configuration does not provide any memory usage guidance, the kernel applies a default strategy to fill memory demands. Our final configuration, called *hot/cold organize*, uses the custom kernel with our custom JVM that divides the application's heap into two separate colored spaces: one space colored orange for hot objects and the other colored blue for cold objects. Using the memory coloring API, we configure our framework so that, whenever possible, demands from different colored spaces are filled with physical pages from trays corresponding to different DIMMs. We also modify HotSpot's object allocator to recognize objects of the ColdObject type and assign them to the space colored blue. Alternatively, HotObjects, as well as any other objects created by the MemBench workload, are allocated to the orange space. Thus, the effect of the hot/cold organize configuration is to ensure that, of the four 8GB DIMMs in our node, three of the DIMMs are occupied with pages that back ColdObjects, while the other DIMM is used for the rest of the MemBench memory, including the HotObjects.

We ran the MemBench benchmark with each of our three configurations and with different delay parameters to vary the bandwidth requirements for the workload. Our results in this section report the average over ten runs for each configuration / delay pair.

Execution of the MemBench benchmark occurs in two phases: the *allocation* phase and the

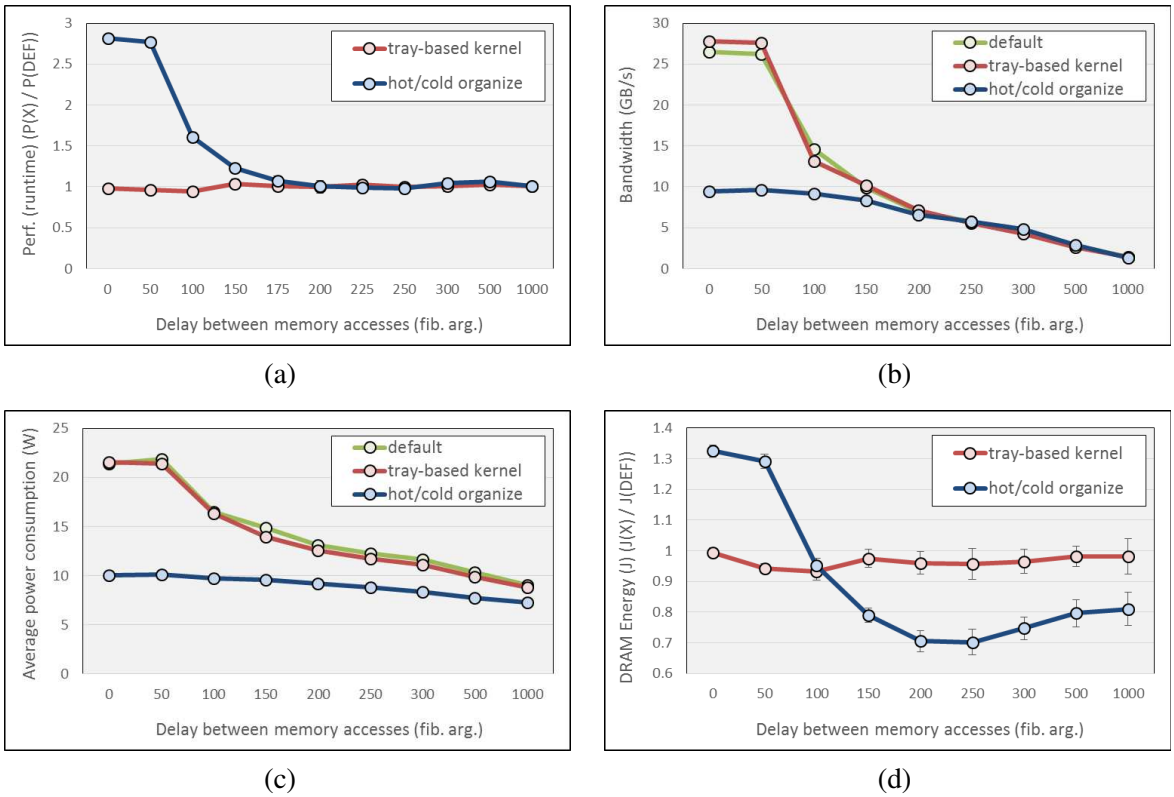


Figure 5.2: Performance (a), bandwidth (b), average DRAM power (c), and DRAM energy (d) for the MemBench benchmark.

scan phase. We first consider the potential slowdown in the allocation phase due to overheads in our custom kernel and JVM framework. Table 5.1 presents the time required to allocate the object array in each configuration. Thus, allocating pages over tray structures with the custom kernel takes about 15 seconds longer (a 16% slowdown) compared to the default kernel. The hot/cold organize configuration incurs an additional slowdown because the JVM is required to distinguish hot and cold objects at each allocation point.

Next, we examine the performance and energy characteristics of the MemBench workload during the scan phase when threads are scanning and accessing the object array. Figure 5.2(a) compares the performance (runtime) of the tray-based kernel and hot/cold organize configurations to the default performance with the same delay factor. To estimate the degree of variability in our performance and energy results, we compute 95% confidence intervals for the difference between the means (Georges et al., 2007), and plot these intervals as error bars in Figures 5.2(a) and

5.2(d). Thus, we can see that the tray-based kernel configuration performs about the same as the default configuration regardless of delay factor. The hot/cold organize configuration exhibits relatively poor performance when there is little delay between memory accesses, but performs more like the default configuration as the delay is increased. Interestingly, we find this observed performance difference is directly related to bandwidth of the memory controller. Figure 5.2(b) shows the average read/write bandwidth (in GB/s) for MemBench with each configuration at varying delays. Notice that the default and tray-based kernel configurations actually produce much higher memory bandwidth than hot/cold organize when the delay factor is low. With very little delay between memory accesses, MemBench requires very high memory bandwidth to achieve good performance. The default and tray-based kernel configurations both distribute hot objects across the system's memory devices, allowing the system to exploit rank and channel parallelism to achieve higher bandwidths. Alternatively, the hot/cold organize configuration co-locates all the hot objects onto a single DIMM that is connected to a single channel. Consequently, this configuration cannot attain the bandwidth required to achieve good performance when the delay is low. Increasing the delay between memory accesses reduces the bandwidth requirements for the workload, and thus, enables hot/cold organize to achieve performance similar to the other configurations.

While co-locating the hot objects onto a single DIMM restricts bandwidth, this organization consumes much less power, on average, than distributing accesses across all the memory devices. Figure 5.2(c) plots the average power consumption (in W) for MemBench with each configuration at varying delays. In the experiments with little delay between memory accesses, hot/cold organize consumes less power, in part, because it cannot handle the high bandwidth requirements, and so, actually accesses memory at a slower rate than the other configurations. However, we find that significant power differences persist even when the bandwidth requirements are reduced. This effect occurs because most of the DIMMs in the hot/cold organize configuration are occupied with objects that are rarely accessed, which allows the memory controller to transition them to low power states more often. Therefore, there is a significant potential for DRAM *energy* savings with this configuration. Figure 5.2(d) compares the total DRAM energy consumed during the entire

program run with the tray-based kernel and hot/cold organize configurations to the DRAM energy consumed by the default performance with the same delay factor. Thus, in experiments with low delay, the hot-cold organize configuration actually requires more DRAM energy than the default configuration due to its relatively poor performance. However, our custom JVM framework enables significant energy savings for workloads that do not require as much memory bandwidth. At a maximum, we achieve 30% DRAM energy savings (with delay=250). We also find that, reducing the workload's bandwidth requirements beyond a certain point (about 5GB/s on our system), results in diminished DRAM energy savings.

5.6 Experimental Framework

In this section, we describe our experimental platform as well as the tools and methodology we use to conduct our experiments, including how we measure power and performance.

5.6.1 Platform

All of the experiments in this paper were run on a single socket of a four-socket Intel S2600CP server system with an Intel Xeon E5-2600 series processor (codename "Sandy Bridge"). The socket we use has 8 2.3GHz cores with hyperthreading enabled (for a total of 16 hardware threads) and 4 8GB DIMMs of Samsung DDR3 memory (part #: M393B1K70CH0-CK0). Each DIMM is connected to its own channel and is comprised of two 4GB ranks. We install 64-bit SUSE Linux Enterprise Server 11 SP1 and select a recent Linux kernel (version 2.6.32) as our default operating system. For all of the experiments in this work, we configure the BIOS to disable interleaving of physical addresses across rank and channel boundaries.

5.6.2 Memory Power and Bandwidth Measurement

We employ Intel's Performance Counter Monitor (PCM) tool (http://www.intel.com/software/pcm, 2012) to estimate DRAM power and bandwidth. PCM is a sampling-based tool that reads various

activity counters (registers) to provide details about how internal resources are used by Intel processors. At every sample, PCM reports an estimate of the DRAM energy consumed (in J) and average read/write bandwidth on the memory controller (in MB/s) since the previous interval. The DRAM power model used by PCM is based on proprietary formulae, and its accuracy depends on the set of DRAM components used. Additionally, the tool estimates the percentage of time each rank resides in a low-power state by counting the number of cycles with the CKE signal de-asserted during the previous interval. During each experiment, we employ PCM to collect samples of DRAM power, bandwidth, and CKE off residencies every two seconds. We aggregate the collected samples during post-processing to provide an estimate of each metric for the entire run.

5.6.3 HotSpot Java Virtual Machine

Oracle's HotSpot Java Virtual Machine (build 1.6.0_24) (Paleczny et al., 2001) provides the base for our JVM implementation. For this work, we used the default HotSpot configuration for server-class applications (http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper 150215.pdf, 2006). This configuration uses the "parallel scavenge" garbage collector (PS-GC), which is a "stop-the-world" collector (i.e. application threads do not run during GC) that is capable of spawning multiple concurrent scavenging threads during GC. Our current implementation makes a few low-level, structural modifications to the HotSpot JVM's object allocation and collection policies to simplify our engineering effort.

HotSpot attempts to automatically configure the size of the heap at startup and dynamically at run-time. After every major collection, the PS-GC collector in HotSpot re-assigns generation boundaries to fit the compacted heap. To avoid having to dynamically re-compute colored space boundaries in our implementation, we configure HotSpot to use a *static heap size* for each experimental run. For each benchmark, we execute a trial run in which we allow HotSpot to configure heap sizes automatically. Then, for each experimental run, we use command line options to set the eden, survivor, and tenured space sizes according to the boundaries used when the heap was at its maximum size during the trial run. We further adjust the young generation sizes to eliminate the

Table 5.2: Benchmarks from SPECjvm2008

Benchmark	Abrv.	Description	GB
compiler.compiler	com	Java compiler (compiles `javac`)	8.92
compiler.sunflow	csf	Java compiler (compiles `sunflow`)	11.15
compress	cmp	Lempel-Ziv compression	18.9
crypto.aes	aes	encrypt / decrypt with AES	23.87
crypto.rsa	rsa	encrypt / decrypt with RSA	1.09
crypto.signverify	svf	sign and verify w/ various protocols	22.61
serial	ser	serialize / deserialize objects	23.95
derby	drb	Java database	14.11
sunflow	sun	graphics visualization	10.23
mpegaudio	mpg	mp3 decoding	14.48
xml.transform	xtr	applies style sheets to XML docs	23.16
xml.validation	xva	validates XML docs	9.26
scimark.fft	fft	Fast Fourier Transform	18.05
scimark.lu	lu	LU matrix factorization	14.61
scimark.sor	sor	Jacobi Successive Over-relaxation	4.28
scimark.sparse	spa	sparse matrix multiply	9.4
scimark.monte_carlo	mco	Monte Carlo to approximate π	0.52

need for major GC's. In this way, our custom JVM avoids resizing the heap during experimental runs.

Additionally, our current implementation disables an object allocation optimization known as thread local allocation buffers (TLABs). HotSpot generates optimized assembly code for TLAB allocations, which speeds up allocations, but makes it difficult to modify the allocation path. Thus, for our current implementation, we disable TLABs and force allocations through a slower path in the VM. Note that these implementation choices are not due to any restriction with our design, but only to alleviate our initial implementation effort. In the future, we will enable heap re-sizing and add TLAB support to our JVM framework.

5.6.4 Benchmarks

We evaluate our framework using all of the benchmarks from the SPECjvm2008 suite of applications (SPEC2008, 2008). Table 5.2 lists each of our benchmark applications, along with an abbreviation for presenting results, a short description, and the peak resident set size (in GB) with our default configuration. The SPECjvm2008 suite employs a harness program to continuously iterate the workload for each run, starting a new operation as soon as a previous operation completes. Each run includes a warmup period of one minute and an iteration period of at least four

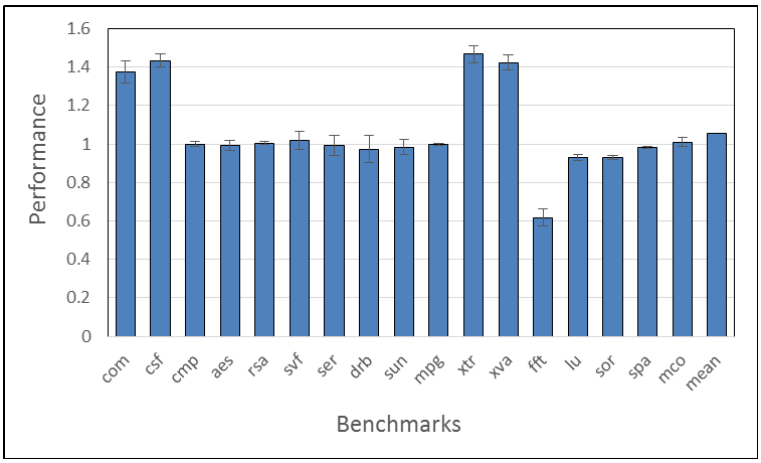


Figure 5.3: Performance of our baseline configuration with the custom kernel compared to the default configuration with the unmodified Linux kernel.

minutes. The score for each run is the average time it takes to complete one operation during the iteration period. For all of our performance and energy measurements, we report the average result of five runs as the final score for the benchmark.

5.6.5 Baseline Configuration Performance

Our framework employs a custom Linux kernel that has been modified to interpret and apply memory usage guidance from the application layer. In our later experiments, we compare our custom JVM configuration to a baseline configuration with the default (unmodified) HotSpot JVM *running on the same custom kernel*. In this way, we are able to directly evaluate the performance and energy effects of our custom JVM framework. It is important to note that the custom kernel configuration we use as our baseline affects the performance of some of our benchmarks. For our baseline configuration, we opt for a default allocation strategy that selects pages from trays in a round-robin fashion. The effect of this strategy is to increase rank and channel parallelism by allocating pages across as many memory devices as possible. Figure 5.3 displays the performance of each benchmark when run with the baseline configuration with the custom kernel compared to a default configuration with an unmodified Linux kernel. Thus, most of the benchmarks exhibit about the same performance with the default kernel as with our custom kernel configuration. However, the

compiler and *xml* pairs of benchmarks show significant performance degradations (of about 40%) when compared to performance with the default kernel. One possible explanation is that these workloads allocate memory objects at a very fast rate, and thus, the overhead required to allocate pages over tray structures is more significant. Additionally, we find that the performance of some benchmarks, most notably *fft*, actually improves with the custom kernel configuration. This effect is likely due to the custom kernel improving rank and channel parallelism with its default allocation policy. On average, the custom kernel configuration performs about 5% worse than the default kernel with our set of benchmarks.

5.7 Automatic Cross-Layer Memory Management

In this section, we describe and evaluate our profiling-based approach for *automatically* partitioning hot and cold objects at runtime.

5.7.1 Profiling for Hot and Cold Allocation Points

We employ a two-step approach for automatically partitioning the application's hot and cold objects. First, we conduct an offline profile run to gather information about how the application allocates and uses objects in memory. We then analyze the memory profiling information to partition the application's allocation points into two sets according to the memory usage activity (hotness or coldness) of the objects created at each point. In the next run, the JVM loads the hot/cold classification of the program's allocation points, and uses it to assign new objects to separate colored spaces.

We first describe our custom framework for profiling memory usage activity related to each program allocation point. Our profiling framework instruments the HotSpot bytecode interpreter to construct a hash table containing information about each program allocation point. The hash table is indexed by program allocation points (identified by method and bytecode index) and values are a list of records describing the size and access count of each object created at that point. During

the run, whenever the VM interprets a `_new` instruction (or one of its variants), it creates a record in the global hash table to indicate that a new object was created at that point. Then, whenever an object is accessed using one of the object reference bytecodes (`getfield`, `putfield`, etc.), the VM increments its access count.

After collecting the application's memory profile, the next step is to assign colors to the allocation points according to the memory usage activity of the objects created at each point. For this analysis, we are interested in maximizing the size of the cold space because a larger space of cold objects ensures that a larger portion of DRAM devices will be able to reside in a low power state. We can express this problem as an instance of the classical 0/1 knapsack optimization problem. In this formulation, each allocation point is an item, which has a value equal to the total size of the objects created at that point, and a weight equal to the sum of the access counts for those objects. We assume some maximum number of object accesses is the capacity of the knapsack. The optimization problem then is to select items (allocation points) such that the combined value (object size) is maximized without the combined weight (access counts) exceeding the capacity of the knapsack. We employ a popular dynamic programming algorithm to approximate a solution to the NP-complete knapsack problem that (nearly) maximize the size of the cold space (Martello & Toth, 1990).

To compare colorings across different workloads, we select knapsack capacities as a percentage of the the total number of accesses in the profile run. For example, with a knapsack capacity of 5%, our technique selects allocation points that account for no more than 5% of the total number of object accesses in the profile run. For the chosen knapsack capacity, we compute a partitioning of the program's allocation points using the approach described above and store the result to a file on disk. Then, at the start of the next program run, the VM loads the partitioning information into a global structure in memory. Whenever the application allocates an object, the VM looks up whether the allocation point has a known color, and if so, assigns the new object to the corresponding colored space. Otherwise, the object is simply assigned to some default colored space (in our experiments, we opt to assign objects created at unknown allocation points to the orange, or hot,

space).

5.7.2 Experimental Evaluation

We conducted a series of experiments to evaluate the performance and effectiveness of our JVM framework with different coloring configurations. For each of the benchmarks in Table 5.2, we collect a profile of the benchmark's memory activity using a small version of the workload. Next, we partition each application's profiled allocation points into different colored sets as described in the previous subsection using knapsack capacities of 2%, 5%, and 10%. We then conduct experiments that use each allocation point partitioning to guide the object coloring in a larger run of each benchmark.

5.7.2.1 Memory Profile Analysis

We use different configurations for our profiling and actual evaluation runs. For SPECjvm2008 benchmarks that are shipped with multiple input sizes (scimark kernels, with the exception of *monte_carlo*), we use the small size for our profile runs, and the large size for the guided runs. Most of our selected benchmarks include only one input size for each application. Fortunately, all our benchmarks are multi-threaded, and the SPECjvm2008 harness scales the workload according to the number of benchmark application threads used in the run. Thus, for our profile runs, we configure the harness to only use one benchmark thread per application. While for the guided runs, the harness automatically selects the number of threads to use based on our machine configuration (which, for our platform, is always at least 16 threads).

Table 5.3 presents the predicted size of cold objects in the profile run and the actual observed size of objects allocated in the cold space during the guided run (normalized by the total size of all objects) for each benchmark and each knapsack coloring. For example, with *compiler.compiler* (*com* in Table 5.3), the 2% knapsack coloring found a set of allocation points that accounts for 14.9% of the total heap size in the profile run (and about 2% of the object accesses), and 14.5% of the total heap size in the larger guided run. We can make a few observations from these results:

Table 5.3: Cold Size / Total Size in profile and guided runs

Bench	2% Knapsack		5% Knapsack		10% Knapsack	
	Profile	Guided	Profile	Guided	Profile	Guided
com	0.149	0.145	0.289	0.301	0.480	0.484
csf	0.152	0.151	0.302	0.291	0.487	0.463
cmp	0.581	0.014	0.811	0.022	0.917	0.458
aes	0.296	0.154	0.531	0.436	0.882	0.859
rsa	0.768	0.413	0.894	0.627	0.960	0.781
svf	0.392	0.040	0.924	0.964	0.982	0.992
ser	0.217	0.193	0.434	0.403	0.686	0.470
drb	0.238	0.411	0.434	0.774	0.528	0.926
sun	0.194	0.078	0.309	0.140	0.340	0.176
mpg	0.946	0.979	0.996	0.995	0.998	0.996
xtr	0.250	0.131	0.468	0.301	0.698	0.458
xva	0.240	0.212	0.475	0.474	0.719	0.632
fft	0.868	0.813	0.996	0.814	0.996	0.814
lu	0.627	0.502	0.719	0.503	0.719	0.503
sor	0.988	0.027	0.990	0.027	0.990	0.027
spa	0.729	0.004	0.737	0.004	0.737	0.005
mco	0.999	0.959	0.999	0.959	0.999	0.958
avg	0.508	0.307	0.665	0.473	0.772	0.588

1. For some benchmarks (e.g. *fft*, *lu*), and *mpg*) a relatively large portion of the objects are very cold in both the profile and guided runs. This result indicates that, if the object access counts are accurate, then there is significant potential to reduce DRAM energy consumption with these workloads because a larger portion of the application's memory can be co-located onto DIMMs that often transition to low power states.

2. Increasing the knapsack capacity increases the cold space size for the profile and guided runs.

3. For several of the benchmarks (e.g. *com*, *mpg*, and *mco*), it seems that the cold space size in the profile run accurately predicts the cold space size in the guided run for several of the benchmarks. However, as expected, for several of the other benchmarks (e.g. *cmp*, *sor*, and *spa*), the cold space size in the profile run does not seem to be correlated with the proportion of cold memory in the guided run.

5.7.3 Controlling the Power State of DRAM Ranks

Table 5.4 shows the CKE off residencies collected for each two second interval and then averaged over the entire program runtime for our baseline configuration. Thus, the CKE off residency values

Table 5.4: Average CKE OFF Residencies: Default Configuration

BM	Channel 0		Channel 1		Channel 2		Channel 3	
	RK0	RK1	RK0	RK1	RK0	RK1	RK0	RK1
com	84	84.11	83.98	84	77.42	84.09	84.2	84.11
csf	84.24	83.87	83.94	83.99	76.83	84.37	84.53	84.46
cmp	65.91	66.08	66.15	66.06	64.08	66.04	65.66	65.55
aes	87.82	87.86	87.8	87.64	87.43	86.05	87.48	87.87
rsa	93.52	93.54	93.07	92.88	92.06	93.51	93.49	93.48
svf	80.74	80.8	81.11	81.43	80.37	79.9	81.35	81.28
ser	91.47	91.46	91.39	91.62	90.82	91.19	91.57	91.87
drb	91.38	91.12	91.11	91.12	92.42	87.97	91.22	91.3
sun	91.78	91.59	91.78	92.29	89.9	92.54	92.15	91.63
mpg	91.36	91.68	92.05	92.04	89.34	91.97	91.89	91.4
xtr	89.87	89.61	89.5	89.29	87.43	88.07	89.63	89.93
xva	88.42	88.2	87.58	87.67	85.63	88.37	88.15	88.46
fft	2.01	1.95	1.96	1.86	1.67	1.93	2.04	1.95
lu	1.76	1.75	1.73	1.71	1.5	1.74	1.81	1.81
sor	1.37	1.34	1.29	1.26	1.13	1.32	1.42	1.39
spa	3.7	3.51	3.37	3.34	2.74	3.56	3.77	3.75
mco	99.82	99.82	99.43	99.1	96.01	97.15	99.45	99.61
avg	67.6	67.55	67.48	67.49	65.69	67.05	67.64	67.64

Table 5.5: Average CKE OFF Residencies: 2% Knapsack Coloring

BM	Cold DIMMs				Hot DIMMs			
	Channel 0		Channel 1		Channel 2		Channel 3	
	RK0	RK1	RK0	RK1	RK0	RK1	RK0	RK1
com	89.75	97.07	98.27	98.23	67.03	73.16	73.22	73.19
csf	87.42	97.2	98.81	98.82	66.92	73.05	73.19	73.35
cmp	97.81	98.01	99.69	99.45	54.56	43.29	48.6	48.57
aes	90.62	97.44	98.03	97.95	81.15	81.47	82.1	82.1
rsa	86.03	96.88	98.76	99.3	92.27	92.9	92.91	92.92
svf	93.18	97.58	100	100	66.83	69.23	68.75	68.75
ser	86.72	97.27	99.36	99.3	81.78	84.2	84.03	83.94
drb	70.59	97.58	98.83	98.94	84.85	73.76	80.1	80.16
sun	93.73	97.58	99.63	99.34	84.56	87.15	87.2	87.24
mpg	91.38	91.87	98.1	98.77	89.45	92.41	92.09	91.8
xtr	86.37	97.58	100	100	76.91	80.17	80.23	80.26
xva	77.37	97.58	100	100	69.53	72.28	72.26	72.37
fft	96.45	98.54	98.2	98.19	0.86	0.92	0.93	0.94
lu	96.71	97.24	99.04	98.64	0.81	0.87	0.86	0.86
sor	98.25	98.25	98.89	99.35	0.29	0.3	0.3	0.29
spa	98.73	98.73	98.66	98.65	0.97	1.05	1.04	1.07
mco	99.67	99.7	100	100	96.44	97.17	98.75	99.27
avg	90.63	97.42	99.07	99.11	59.72	60.2	60.97	61

show the percentage of runtime each DRAM rank in our machine was in a low power state during the program run. We find that many of the benchmarks in our set do not access memory very frequently. For these workloads (such as *monte_carlo*), the memory controller is able to transition the ranks to low power states for most ($> 90\%$) of the benchmark run. On the other hand, high performance scientific workloads such as the scimark kernels keep the memory ranks in an active state for almost the entire run, and thus consume a large amount of energy. Also note that memory usage activity is distributed evenly across the ranks. This is an effect of the baseline configuration's physical page allocation policy, which attempts to distribute allocations across ranks to increase rank and channel parallelism.

Tables 5.5, 5.6, and 5.7 show the same information as Table 5.4, but for our guided runs with knapsack capacities 2%, 5%, and 10%, respectively. For each of our guided runs, we configure

Table 5.6: Average CKE OFF Residencies: 5% Knapsack Coloring

BM	Cold DIMMs				Hot DIMMs			
	Channel 0		Channel 1		Channel 2		Channel 3	
	RK0	RK1	RK0	RK1	RK0	RK1	RK0	RK1
com	76.7	97.58	100	100	63.11	69.84	69.79	69.78
csf	73.46	97.2	98.74	98.85	63.56	70.68	70.73	70.47
cmp	39.41	97.39	100	100	58.3	59.06	59.27	58.93
aes	91.14	90.98	97.69	99.5	83.62	83.16	83.84	83.78
rsa	84.03	97.58	99	98.31	92.8	94.08	94.47	94.48
svf	67.98	66.5	95.55	97.34	92.31	96.57	96.59	96.58
ser	71.82	96.98	100	100	79	82.03	82.02	82.08
drb	73.4	93	98.57	99.01	93.45	87.53	91.23	91.35
sun	90.47	97.2	98.74	98.74	84.4	87.11	87.33	87.49
mpg	88.63	89.41	95.09	97.3	92.21	96.3	96.34	96.32
xtr	63.02	97.01	98.01	98	67.84	73.09	73.13	73.1
xva	41.25	96.92	99.15	99.52	69.2	73.21	73.19	73.01
fft	94.76	96.96	99.88	99.93	0.87	0.93	0.93	0.93
lu	96.68	97.36	99.32	99.4	0.91	0.96	0.95	0.96
sor	99.86	99.89	99.52	99.14	0.79	0.81	0.81	0.81
spa	98.9	99.05	100	100	0.71	0.78	0.77	0.79
mco	97.66	98.05	100	100	97.85	98.61	98.98	98.27
avg	79.36	94.65	98.78	99.12	61.23	63.22	63.55	63.48

Table 5.7: Average CKE OFF Residencies: 10% Knapsack Coloring

BM	Cold DIMMs				Hot DIMMs			
	Channel 0		Channel 1		Channel 2		Channel 3	
	RK0	RK1	RK0	RK1	RK0	RK1	RK0	RK1
com	59.15	81.41	97.77	97.9	50.62	59.48	59.63	59.61
csf	48.57	84.66	98.43	98.25	50.76	60.28	60.68	60.63
cmp	35.16	81.24	66.15	97.87	77.4	79.87	80.06	79.88
aes	87.26	85.82	88.67	98.97	93.36	95.53	95.63	94.96
rsa	82.91	97.58	100	100	94.66	94.92	94.99	95.27
svf	67.72	66.98	95.88	96.89	92.56	97.18	97.25	97.25
ser	73.26	96.43	99.09	99	82.29	85.36	85.33	85.04
drb	74.03	93.26	99.21	99.49	95.32	90.3	93.75	93.89
sun	88.22	97.58	99.33	99.1	84.62	86.98	86.86	87.21
mpg	87.67	87.57	97.57	98.36	92.92	96.6	96.62	96.31
xtr	64.85	81.37	100	100	68.79	74.51	74.53	74.16
xva	39.94	96.54	100	100	76.21	80.06	80.05	80.54
fft	95.55	98.17	100	100	0.8	0.86	0.86	0.87
lu	96.45	97.27	98.56	99.03	0.61	0.65	0.67	0.67
sor	98.5	98.47	99.27	99.58	0.37	0.38	0.39	0.38
spa	98.71	98.72	98.66	98.65	1.18	1.27	1.27	1.3
mco	98.07	98.84	100	100	97.44	98.63	98.14	97.78
avg	76.24	90.7	96.39	99.01	62.35	64.87	65.1	65.04

our framework to co-locate pages from the hot and cold spaces onto different sets of DIMMs. Thus, Tables 5.5–5.7 contain an extra row to distinguish ranks that are used for hot objects from ranks that are used for cold objects. In addition to controlling object placement, our framework enables each colored space to select its own allocation strategy to maximize efficiency. The cold space strategy attempts to only select pages from trays that have already furnished another page for similar use. In this way, the strategy for the cold space attempts to minimize the number of memory devices that need to stay powered up. The hot space uses the same strategy we use for our baseline configuration, only applied over a smaller set of devices.

We find that the cold space allocation strategy is very effective at keeping most of the cold ranks in a low power state. This result is expected because most of the guided runs do not allocate enough memory in the cold space to affect more than two ranks. The accuracy of our framework

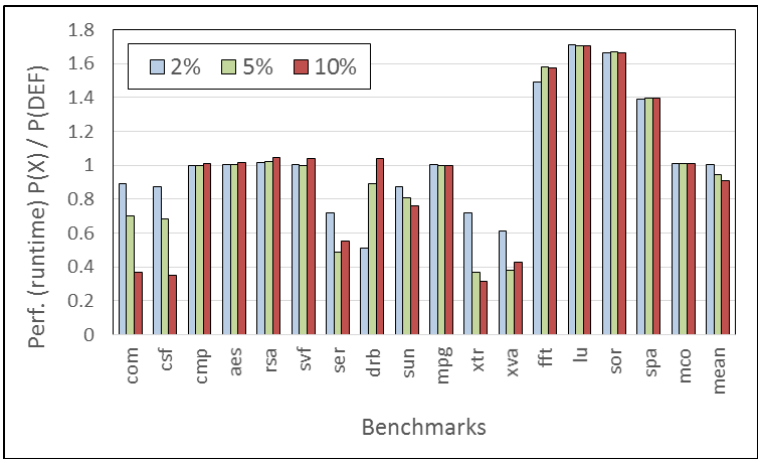


Figure 5.4: Perf. with each colored configuration compared to default.

for distinguishing hot and cold objects is more mixed. There are some benchmarks, such as *fft* and *lu*, that contain a set of very hot objects, and the guided run is able to distinguish these objects from the majority of memory, which is actually cold. On the other hand, guided runs of some of the other benchmarks, such as *svf* and *mpg*, have ranks used by the cold space that reside in a low-power state less often than ranks used by the hot space. Despite this result, the overall trend shows that guided runs are effective at distinguishing hot and cold objects. There is also a clear trend of increasing cold space memory activity as the capacity for the knapsack coloring used to guide the run is increased.

5.7.4 Performance and Energy Evaluation

Figures 5.4 and 5.5 compare the performance and DRAM energy consumption of our guided run configurations to our baseline for all benchmarks. Similar to our approach for Figures 5.2(a) and 5.2(d), we compute 95% confidence intervals for the difference between the means to estimate the degree of variability in these results. We find that variability is low for both sets of results. The maximum length interval for performance is $\pm$7.4% and for energy $\pm$6.7% (both for *derby*), while the vast majority of intervals are $< \pm 3\%$.

As expected, the performance of some of the workloads with the guided run configurations, such as the scimark benchmarks, degrades when compared to the default configuration. The base-

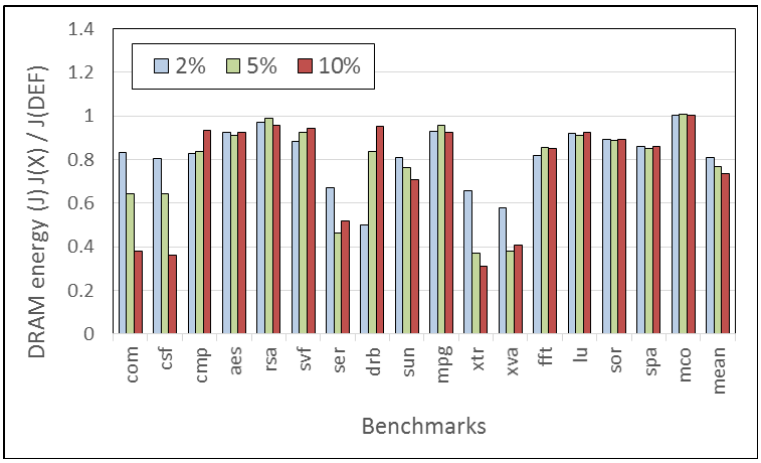


Figure 5.5: DRAM energy consumed with each colored configuration compared to default.

line configuration enables more rank and channel parallelism than the guided run configurations, and thus, workloads with high bandwidth requirements do not perform as well with the guided runs. In the worst case (*lu*), this effect causes a performance loss of over 70%. Surprisingly, several of the SPECjvm2008 benchmarks actually *improve* performance with the custom JVM framework. We believe this effect is caused by more favorable object location (on a page) and page placement produced by the coloring scheme, which has been reported to have a significant impact on performance (Hirzel et al., 2003; Guyer & McKinley, 2004; Hirzel, 2007).

Most significantly, we find that our approach is able to substantially reduce DRAM energy savings over the baseline. On average, the guided run configurations produce DRAM energy savings of 19%, 23%, and 26% for configurations with knapsack capacities of 2%, 5%, and 10%, respectively. While some large energy savings correspond to guided runs that also show large performance improvements, workloads that degrade performance also yield significant DRAM energy savings with our guided run configurations.

5.8 Future Work

Our immediate future plan is to make our framework compatible with HotSpot's thread-local allocation buffers and heap resizing. We have already implemented mechanisms to replace our existing

offline profiling approach with *online* profiling in the HotSpot VM. An online profiling scheme is advantageous since it allows the objects to adaptively move between heap partitions in response to program phase changes. Unfortunately, our online profiling scheme currently involves very high run-time overhead. In the future, we will explore ways to make online profiling schemes practical. Our experiments in this work revealed the importance of *appropriate* data object location on a page, and placement of a page in physical memory. We plan to explore and effectively resolve this issue in the future.

5.9 Conclusion

Energy efficiency has become a first-class constraint in a wide range of computing domains. We contend that energy efficiency cannot be achieved by either the operating system (OS) or the architecture acting alone, but needs guidance from the application and communication with the hardware to ideally allocate and recycle physical memory. In this work, we design the first application level framework to automatically partition program memory based on profiled data object usage patterns, which is then communicated to the OS. Existing OS frameworks can then use this guidance to decide virtual–physical page mappings.

We use our framework to show that memory power saving is correlated with the program's bandwidth requirement and performance. Our results indicate that a significant portion of the memory allocated by many programs is *cold*. Co-allocating such cold objects on a single memory module can result in significant energy savings. Overall, our cross-layer framework shows good potential to reduce DRAM energy usage, which may come at the cost of performance for programs with high bandwidth requirements.

Chapter 6

Conclusion

Although managed languages have been widely popular for more than a decade, researchers continue to explore the opportunities and challenges posed by virtual machine architectures to improve program efficiency. In this dissertation, we conduct a series of detailed studies using the industry-standard HotSpot Java Virtual Machine to extend and advance these efforts. Our first study extensively explores single-tier and multi-tier JIT compilation policies in order to find strategies that realize the best program performance for existing and future machines. We find that the most effective JIT compilation strategy depends on several factors, such as the availability of free computing resources and the compiling speed and quality of generated code by the compiler(s) employed, and we provide policy recommendations for available single/multi-core and future many-core machines. Our next study constructs a robust, production-quality framework for dynamic compiler phase selection exploration. Using this framework, we find that, customized phase selections can significantly improve program performance in many cases, but current heuristics are unlikely to attain these benefits. Our third study describes the design and implementation of an innovative memory management framework that enables applications to provide guidance to the operating system during memory management. In this work, we empirically demonstrate that application guidance can be used to achieve various objectives, including power savings, capacity provisioning, and performance optimization. Our final study integrates the application guidance framework

with the HotSpot JVM to provide automatic cross-layer memory management, and presents detailed experimental and performance analysis that demonstrates the energy-saving potential of this approach.

References

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., & Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization* (pp. 295–305). Washington, DC, USA: IEEE Computer Society.

Almagor, L., Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S. W., Subramanian, D., Torczon, L., & Waterman, T. (2004). Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (pp. 231–239). New York, NY, USA: ACM Press.

Anagnostopoulou, V., Dimitrov, M., & Doshi, K. A. (2012). Sla-guided energy savings for enterprise servers. In *IEEE International Symposium on Performance Analysis of Systems and Software* (pp. 120–121).

Arnold, M., Fink, S., Grove, D., Hind, M., & Sweeney, P. F. (2000a). Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 47–65).

Arnold, M., Fink, S., Grove, D., Hind, M., & Sweeney, P. F. (2000b). Adaptive optimization in the Jalapeño JVM: The controller's analytical model. In Proceedings of the 3rd ACM Workshop on Feedback Directed and Dynamic Optimization (FDDO '00).

Arnold, M., Fink, S., Grove, D., Hind, M., & Sweeney, P. F. (2005). A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2), 449–466.

Arnold, M., Hind, M., & Ryder, B. G. (2002). Online feedback-directed optimization of Java. *SIGPLAN Not.*, 37(11), 111–129.

Banga, G., Druschel, P., & Mogul, J. C. (1999). Resource containers: a new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99 (pp. 45–58).: USENIX Association.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., & Kozyrakis, C. (2012). Dune: safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12 (pp. 335–348).: USENIX Association.

Bi, M., Duan, R., & Gniady, C. (2010). Delay-Hiding energy management mechanisms for DRAM. In *International Symposium on High Performance Computer Architecture* (pp. 1–10).

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford University Press, Inc.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., & Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06 (pp. 169–190).

Böhm, I., Edler von Koch, T. J., Kyle, S. C., Franke, B., & Topham, N. (2011). Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11 (pp. 74–85).

Brown, A. D. & Mowry, T. C. (2000). Taming the memory hogs: using compiler-inserted releases

to manage physical memory intelligently. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00: USENIX Association.

Bruening, D. & Duesterwald, E. (2000). Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization* (pp. 13–20).

Cavazos, J. & O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 229–240). New York, NY, USA: ACM.

Chang, P. P., Mahlke, S. A., & mei W. Hwu, W. (1991). Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21, 1301–1321.

Chow, K. & Wu, Y. (1999). Feedback-directed selection and characterization of compiler optimizatons. Proc. 2nd Workshop on Feedback Directed Optimization.

Cooper, K. D., Schielke, P. J., & Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems* (pp. 1–9).

Corbet, J. (2010). Memory Compaction.

David, H., Fallin, C., Gorbatov, E., Hanebutte, U. R., & Mutlu, O. (2011). Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11 (pp. 31–40).

Delaluz, V., Sivasubramaniam, A., Kandemir, M., Vijaykrishnan, N., & Irwin, M. (2002). Scheduler-based dram energy management. In *Design Automation Conference* (pp. 697–702).

Deutsch, L. P. & Schiffman, A. M. (1984). Efficient implementation of the smalltalk-80 system.

In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 297–302). New York, NY, USA: ACM.

Engler, D. R., Kaashoek, M. F., & O'Toole, Jr., J. (1995). Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5), 251–266.

Esmaeilzadeh, H., Cao, T., Xi, Y., Blackburn, S. M., & McKinley, K. S. (2011). Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI (pp. 319–332). New York, NY, USA: ACM.

Fan, X., Ellis, C., & Lebeck, A. (2003). Modeling of dram power control policies using deterministic and stochastic petri nets. In *Power-Aware Computer Systems* (pp. 37–41).

Fursin, G., Kashnikov, Y., Memon, A., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C., & O'Boyle, M. (2011). Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39, 296–327.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications*, OOPSLA '07 (pp. 57–76).

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st edition.

Gonzalez, A. (2010). Android Linux Kernel Additions.

Gosling, J., Joy, B., Steele, G., & Bracha, G. (2005). *The Java(TM) Language Specification (3rd Edition)*. Prentice Hall, third edition.

Graham, S. L., Kessler, P. B., & Mckusick, M. K. (1982). Gprof: A call graph execution profiler. *SIGPLAN Notices*, 17(6), 120–126.

Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M., & Sundaresan, V. (2004). Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the conference on Virtual Machine Research And Technology Symposium* (pp.1̃2).

Gu, D. & Verbrugge, C. (2008). Phase-based adaptive recompilation in a jvm. In *Proceedings of the 6th IEEE/ACM symposium on Code generation and optimization*, CGO '08 (pp. 24–34).

Guyer, S. Z. & McKinley, K. S. (2004). Finding your cronies: Static analysis for dynamic object colocation. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04 (pp. 237–250).

Haneda, M., Knijnenburg, P. M. W., & Wijshoff, H. A. G. (2005a). Generating new general compiler optimization settings. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing* (pp. 161–168).

Haneda, M., Knijnenburg, P. M. W., & Wijshoff, H. A. G. (2005b). Optimizing general purpose compiler optimization. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05 (pp. 180–188). New York, NY, USA: ACM.

Hansen, G. J. (1974). *Adaptive systems for the dynamic run-time optimization of programs*. PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA.

Harris, T. (1998). Controlling run-time compilation. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (pp. 75–84).

Hazelwood, K. & Grove, D. (2003). Adaptive online context-sensitive inlining. In *CGO '03: Proceedings of the international symposium on Code generation and optimization* (pp. 253–264). Washington, DC, USA: IEEE Computer Society.

Hirzel, M. (2007). Data layouts for object-oriented programs. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07 (pp. 265–276).

Hirzel, M., Diwan, A., & Hertz, M. (2003). Connectivity-based garbage collection. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03 (pp. 359–373).

Hölzle, U. & Ungar, D. (1996). Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Language Systems*, 18(4), 355–400.

Hoste, K. & Eeckhout, L. (2008). Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08 (pp. 165–174). New York, NY, USA.

http://lwn.net/Articles/445045/ (2011). Ankita Garg: Linux VM Infrastructure to support Memory Power Management.

http://www.acpi.info/spec.htm (2011). Advanced Configuration and Power Interface Specification.

http://www.intel.com/software/pcm (2012). Intel Performance Counter Monitor.

http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper 150215.pdf (2006). Memory Management in the Java HotSpot Virtual Machine.

Huang, H., Pillai, P., & Shin, K. (2003). Design and Implementation of Power-aware Virtual Memory. In *USENIX Annual Technical Conference*.

Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H., & Nakatani, T. (1999). Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99 (pp. 119–128).

Ishizaki, K., Takeuchi, M., Kawachiya, K., Suganuma, T., Gohda, O., Inagaki, T., Koseki, A., Ogata, K., Kawahito, M., Yasue, T., Ogasawara, T., Onodera, T., Komatsu, H., & Nakatani, T. (2003). Effectiveness of cross-platform optimizations for a java just-in-time compiler. In

Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (pp. 187–204).

Jula, A. & Rauchwerger, L. (2009). Two memory allocators that use hints to improve locality. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09 (pp. 109–118).

Kessler, R. E. & Hill, M. D. (1992). Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4), 338–359.

Kleen, A. (2004). A numa api for linux. *SUSE Labs white paper*.

Knuth, D. E. (1971). An empirical study of fortran programs. *Software: Practice and Experience*, 1(2), 105–133.

Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., & Cox, D. (2008). Design of the Java hotspot^{TM}client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1), 1–32.

Krintz, C. (2003). Coupling on-line and off-line profile information to improve program performance. In *CGO '03: Proceedings of the international symposium on Code generation and optimization* (pp. 69–78). Washington, DC, USA.

Krintz, C. & Calder, B. (2001). Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (pp. 156–167).

Krintz, C., Grove, D., Sarkar, V., & Calder, B. (2000). Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8), 717–738.

Kulkarni, P., Arnold, M., & Hind, M. (2007a). Dynamic compilation: the benefits of early investing. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments* (pp. 94–104).

Kulkarni, P. A. (2011). Jit compilation policy for modern machines. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11 (pp. 773–788).

Kulkarni, P. A. & Fuller, J. (2011). Jit compilation policy on single-core and multi-core machines. In *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on* (pp. 54–62).

Kulkarni, P. A., Jantz, M. R., & Whalley, D. B. (2010). Improving both the performance benefits and speed of optimization phase sequence searches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10 (pp. 95–104).

Kulkarni, P. A., Whalley, D. B., & Tyson, G. S. (2007b). Evaluating heuristic optimization phase order search algorithms. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization* (pp. 157–169). Washington, DC, USA: IEEE Computer Society.

Kulkarni, S. & Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12 (pp. 147–162).: ACM.

Lebeck, A. R., Fan, X., Zeng, H., & Ellis, C. (2000). Power aware page allocation. *ACM SIGOPS Operating Systems Review*, 34(5), 105–116.

Lee, H., von Dincklage, D., Diwan, A., & Moss, J. E. B. (2006). Understanding the behavior of compiler optimizations. *Software Practice & Experience*, 36(8), 835–844.

Lefurgy, C., Rajamani, K., Rawson, F., Felter, W., Kistler, M., & Keller, T. W. (2003). Energy management for commercial servers. *Computer*, 36(12), 39–48.

Lin, C.-H., Yang, C.-L., & King, K.-J. (2009). Ppt: joint performance/power/thermal management

of dram memory for multi-core systems. In *ACM/IEEE International Symposium on Low Power Electronics and Design* (pp. 93–98).

Lu, L., Varman, P., & Doshi, K. (2011). Decomposing workload bursts for efficient storage resource management. *IEEE Transactions on Parallel and Distributed Systems*, 22(5), 860 –873.

Magenheimer, D., Mason, C., McCracken, D., & Hackel, K. (2009). Transcendent memory and linux. In *Ottawa Linux Symposium* (pp. 191–200).

Martello, S. & Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc.

Microsoft (2001). *Microsoft C# Language Specifications*. Microsoft Press, first edition.

Namjoshi, M. A. & Kulkarni, P. A. (2010). Novel online profiling for virtual machines. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (pp. 133–144).

Paleczny, M., Vick, C., & Click, C. (2001). The Java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium* (pp. 1–12). Berkeley, CA, USA: USENIX Association.

Pan, Z. & Eigenmann, R. (2008). PEAK: a fast and effective performance tuning system via compiler optimization orchestration. *ACM Trans. Program. Lang. Syst.*, 30, 17:1–17:43.

Petrov, P. & Orailoglu, A. (2003). Virtual page tag reduction for low-power tlbs. In *IEEE International Conference on Computer Design* (pp. 371–374).

Sanchez, R., Amaral, J., Szafron, D., Pirvu, M., & Stoodley, M. (2011). Using machines to learn method-specific compilation strategies. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on* (pp. 257 –266).

Smith, J. & Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

SPEC2008 (2008). Specjvm2008 benchmarks. http://www.spec.org/jvm2008/.

SPEC98 (1998). Specjvm98 benchmarks. http://www.spec.org/jvm98/.

Sundaresan, V., Maier, D., Ramarao, P., & Stoodley, M. (2006). Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06 (pp. 87–97).

Triantafyllis, S., Vachharajani, M., Vachharajani N., & August, D. I. (2003). Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization* (pp. 204–215).: IEEE Computer Society.

Velasco, J. M., Atienza, D., & Olcoz, K. (2012). Memory power optimization of java-based embedded systems exploiting garbage collection information. *Journal of Systems Architecture - Embedded Systems Design*, 58(2), 61–72.

Wang, H., Doshi, K., & Varman, P. (2012). Nested qos: Adaptive burst decomposition for slo guarantees in virtualized servers. *Intel Technology Journal*.

Whitfield, D. L. & Soffa, M. L. (1997). An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6), 1053–1084.

Zhang, C. & Hirzel, M. (2008). Online phase-adaptive data layout selection. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08 (pp. 309–334).

Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., & Kumar, S. (2004). Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS XI (pp. 177–188).: ACM.